パズルゲーム アルゴリズム マニアックス

PuzzleGame Algorithm Maniax

松浦健一郎／司 ゆき 著

SoftBank Creative

パズルゲーム
アルゴリズム
マニアックス

PuzzleGame Algorithm Maniax

松浦健一郎／司 ゆき 著

本書に関する情報をインターネットでも公開しています。
以下のURLよりアクセスしてください。
http://www.sbpnet.jp/books/

はじめに

本書は「パズルゲームの仕組み」を解説する本です。荷物を運ぶゲームや落ち物パズルゲームといった、各種のアクションパズルゲームを扱っています。ゲームプログラミングの入門書としてももちろん使えますが、普通の入門書とは少し毛色が違います。本書は次のような目的にお勧めです。

- パズルゲームの中身がどんな仕組みで動いているのかを知って、ゲームをもっと楽しみたい
- パズルゲームを自分で作りたい
- 専門学校の課題でパズルゲームを作ることになったが、いったいどこから手をつけていいのかわからない
- ブロックやボールを落としたり、積んだり、消したりといったいろいろな動きの実現方法が知りたい
- パズルゲームの「技術カタログ」や「ネタ集」を求めている
- パズルゲームについて熱く語るための、叩き台になる本がほしい
- 昔遊んだり作ったりしたパズルゲームのことを思いだしながら、懐かしい気持ちにひたりたい
- とにかくパズルゲームが好きだ

本書はいわゆる入門書ではないので、どこから読んでもかまいませんし、好きなところだけ拾い読みしても大丈夫です。図解を中心にしているので、まずは絵だけでもパラパラと、漫画を読むような気楽な気分でお読みください。一方で、各技法をシンプルにまとめたプログラムも多数掲載しているので、パズルゲームの制作にも役立てていただけることと思います。

プレイヤーの方にとってもゲームデザイナーの方にとっても、本書がパズルゲームの世界をより深く楽しむためのガイドブックになれば幸いです。

2008年夏

松浦 健一郎／司 ゆき

Contents

Stage 00 序章 Introduction

Stage 01 動かす Move

Stage 02 落とす Drop

Stage 03 つなぐ Connect

Stage 04 ブロック Block

Stage 05 ボール Ball

Stage 06 その他 Others

Appendix

本書の付録CD-ROMには、本書内で紹介するサンプルプログラムのプロジェクトファイルならびに実行ファイルが収録されています。サンプルプログラムは、付録CD-ROMの「Puzzle¥Release」あるいは「Puzzle¥Debug」フォルダ内にある「Puzzle.exe」から実行することができます。
サンプルプログラムの実行には、「Visual C++2008再頒布可能パッケージ（x86）」「DirectXエンドユーザーランタイム（June 2008）」ならびに、DirectX9に対応したビデオカードが必要になります。なお、サンプルプログラムはWindows Vista/XP/2000に対応しています。
サンプルプログラムの実行方法ならびに操作方法については、6ページをご参照ください。

Stage

00

序章

Introduction

荷物を押して目的地に運んだり、ブロックやボールを積み上げたり。あるいは、線路やパイプをつないだり、線で囲んで塗りつぶしたり。ほとんどのゲームは「8方向レバー＋数個のボタン」というシンプルな操作系ながらも、作品によって千差万別のゲーム性を備えていることが、パズルゲームの魅力です。
ハードウェアの進化にともなってグラフィックやサウンドの品質は向上しましたが、たとえ絵や音が簡素でも、パズルゲームの面白さは変わりません。PCやゲーム専用機のように高性能な環境だけではなく、携帯電話や携帯ゲーム機、あるいはFlashなどのWebゲームのように手軽な環境でも、さまざまなパズルゲームを楽しむことができます。

パズルゲームの基本構成

パズルゲームの形式は作品によってさまざまですが、いくつかの典型的なタイプがあります。例えば、迷路を使うタイプのゲームは、次のような要素から構成されています（Fig. 0-1）。

◯ キャラクター

プレイヤーが操作するキャラクターです。人間やロボットのように、人の形をしていることが多いですが、動物や車のこともあります。

◯ カーソル

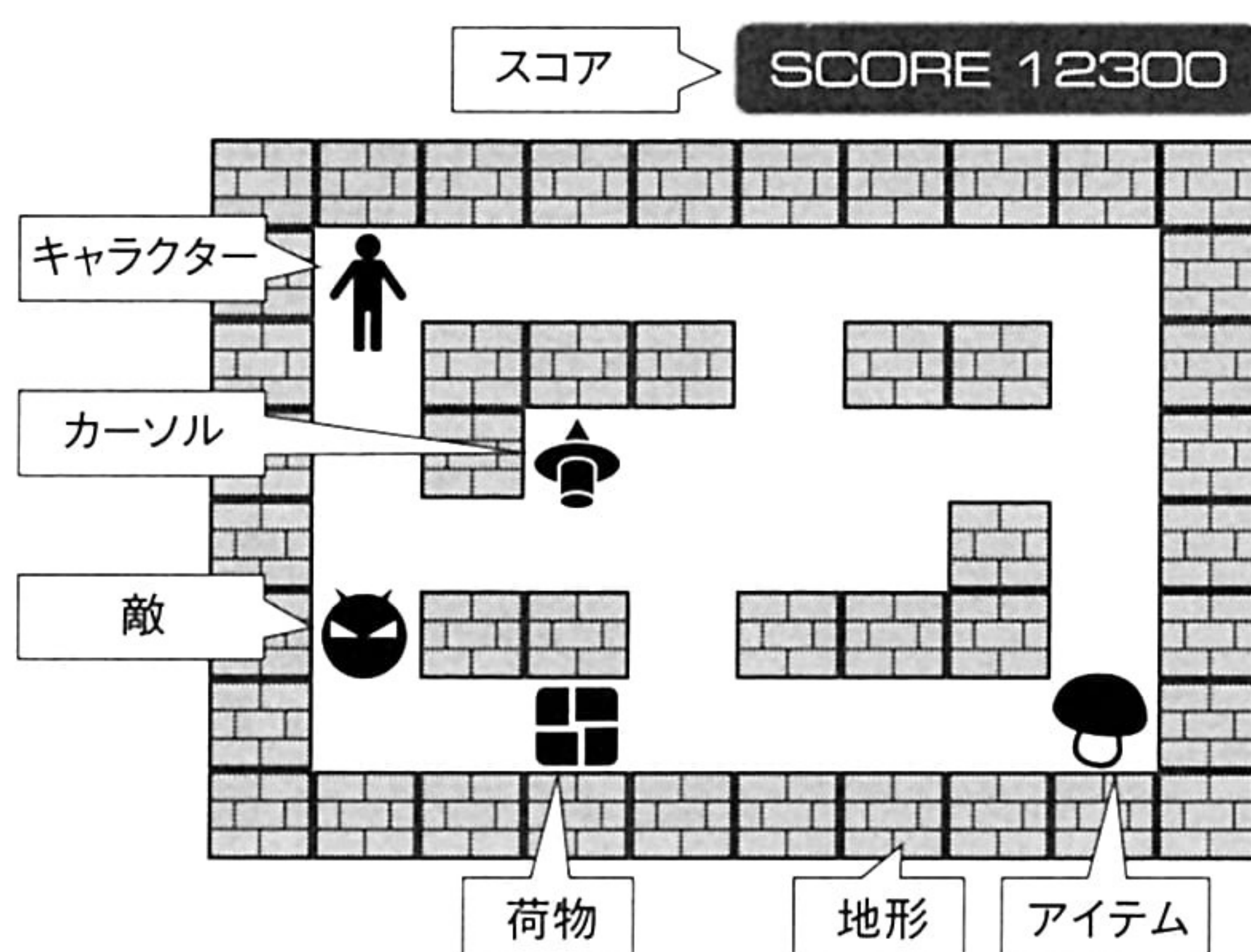
プレイヤーが操作する照準です。対象にカーソルを合わせて、ボタンを押すと、なんらかのアクションを行うことができます。パズルゲームには、キャラクターを使うゲームと、カーソルを使うゲームとがあります。

◯ 敵

キャラクターに攻撃をしかけてくる相手です。多くのゲームでは、敵に触れるとミスになります。謎解きを重視したゲームでは、敵が登場しないこともあります。

◯ 荷物

キャラクターで押したり引いたりして運ぶことができる荷物や岩などです。荷物を使って、敵を潰して倒せるゲームもあります。

Fig. 0-1 迷路を使ったゲームの基本構成

◯ 地形

ステージを構成する床や壁などです。一般にキャラクターや荷物は、地形を通り抜けることができません。

◯ アイテム

キャラクターが拾うことができます。ステージに配置されたアイテムを回収することを、目的としたゲームもあります。

◯ スコア

得点です。敵を倒したり、アイテムを拾ったりすると、スコアが入ります。ステージをクリアすることを重視したゲームでは、スコアがないこともあります。

※

一方、ボールやブロックを積み上げるタイプのゲームは、次のような要素から構成されています（Fig. 0-2）。このタイプのゲームは「落ち物パズルゲーム」と呼ばれることがあります。

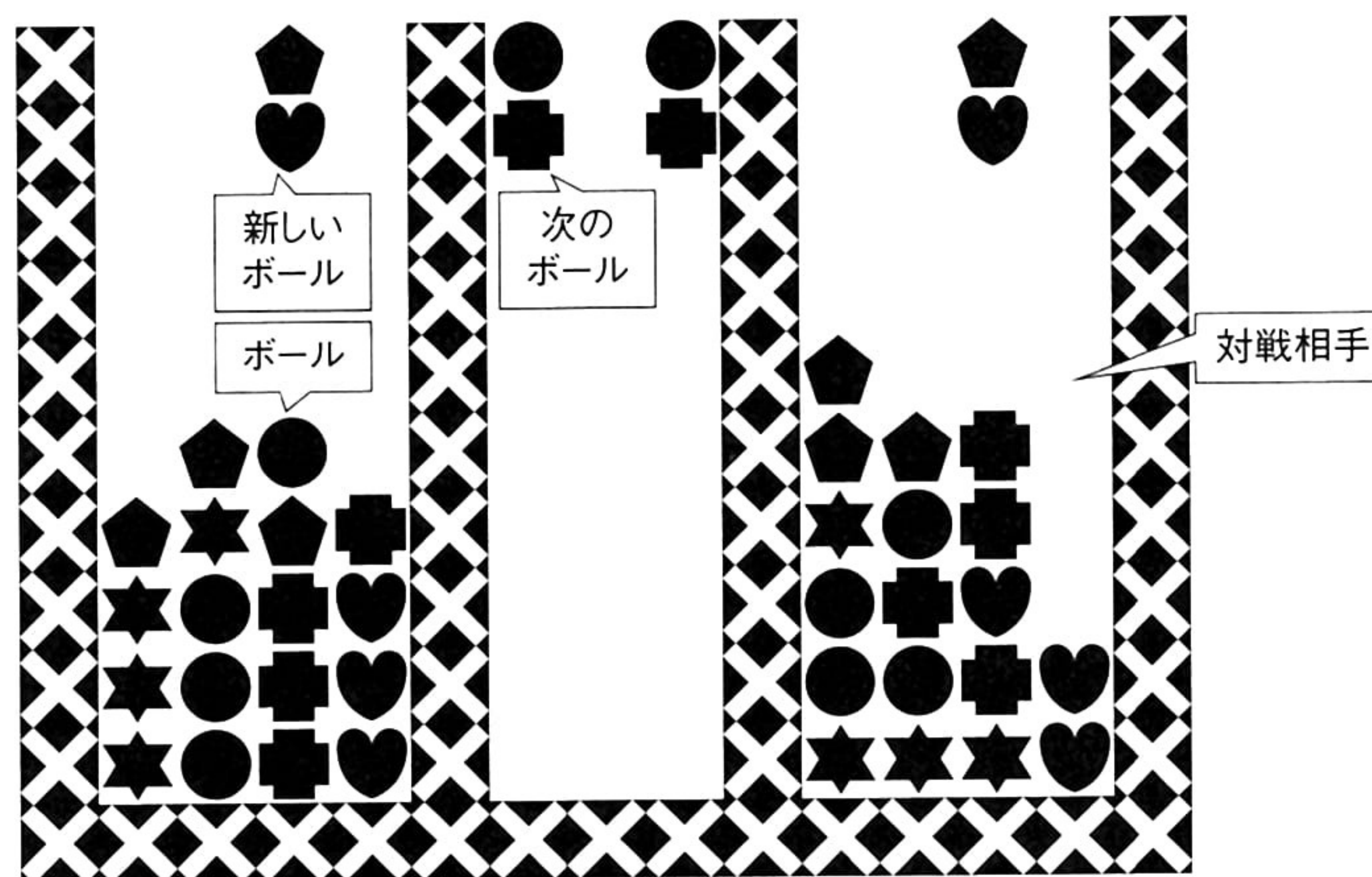
◯ ボール

ステージに積み上げたボールやブロックです。規定数以上の同じ種類のボールやブロックを、上下左右に隣接させたり、縦横斜めに並べたりすると、消すことができます。

◯ 新しいボール

ステージ上方から落ちてくるボールです。プレイヤーがレバーやボタンで操作して、好きな位置に積み上げることができます。

Fig. 0-2 ボールを使ったゲームの基本構成

新しい ボール
次の ボール
ボール
対戦相手

◯ 次のボール

次に出現する新しいボールです。次のボールが示されていると、プレイヤーは先のことを考えながら戦略を練ることができます。

◯ 対戦相手

対戦型のゲームには、対戦相手の領域があります。相手はCPU（コンピュータ）か、他のプレイヤーです。

※

迷路を使ったゲームと、ボールやブロックを積み上げるゲームは、パズルゲームのなかでも特に多いタイプです。その他、独特のルールや画面構成を持つパズルゲームが数多くあります。本書でも、いろんなタイプのゲームについて解説していきます（Fig. 0-3〜0-6）。

Fig. 0-3 線路をつなぐゲーム

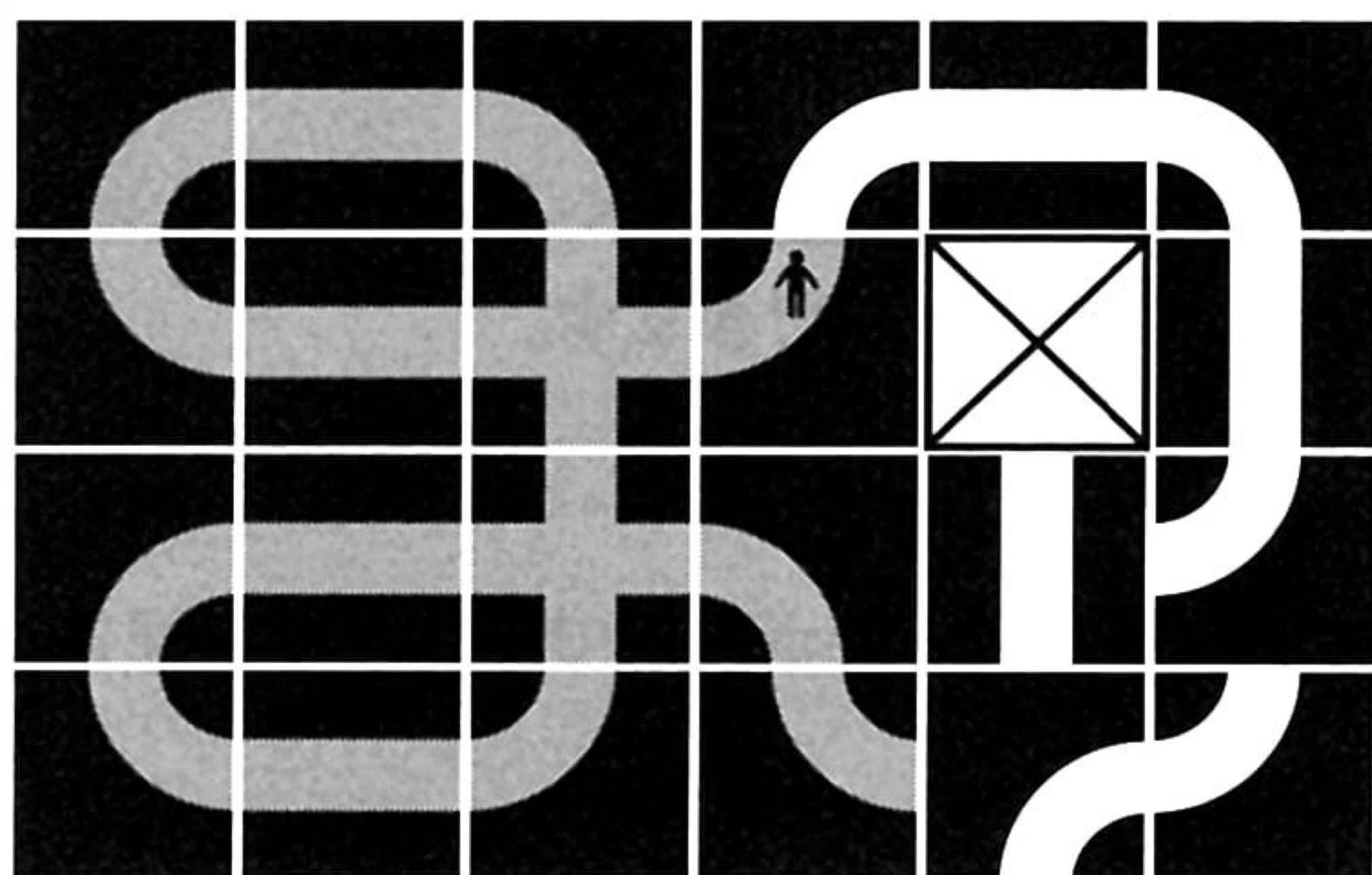



Fig. 0-4 線で囲んで塗りつぶすゲーム

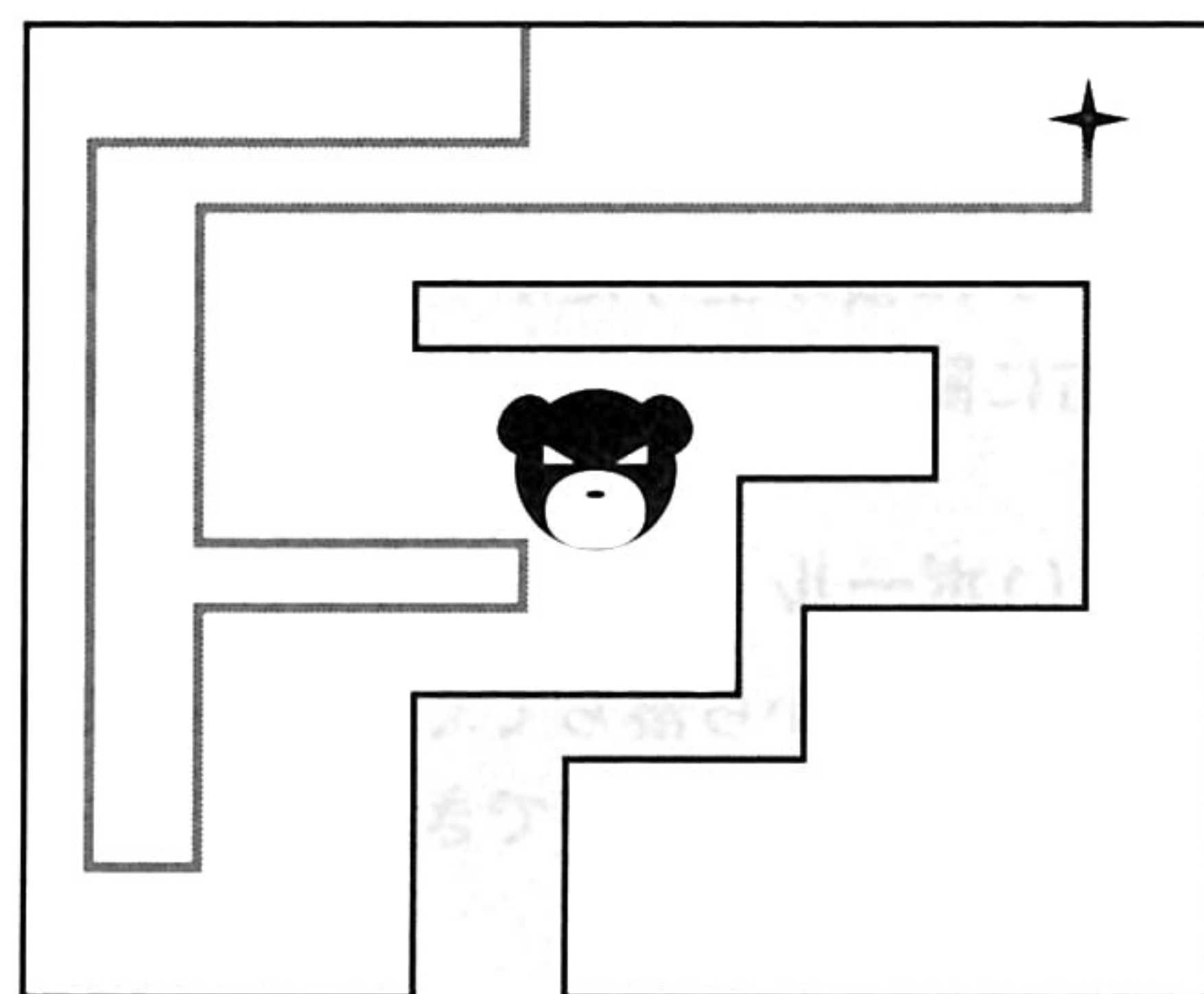



Fig. 0-5 床をマークしてブロックを消すゲーム

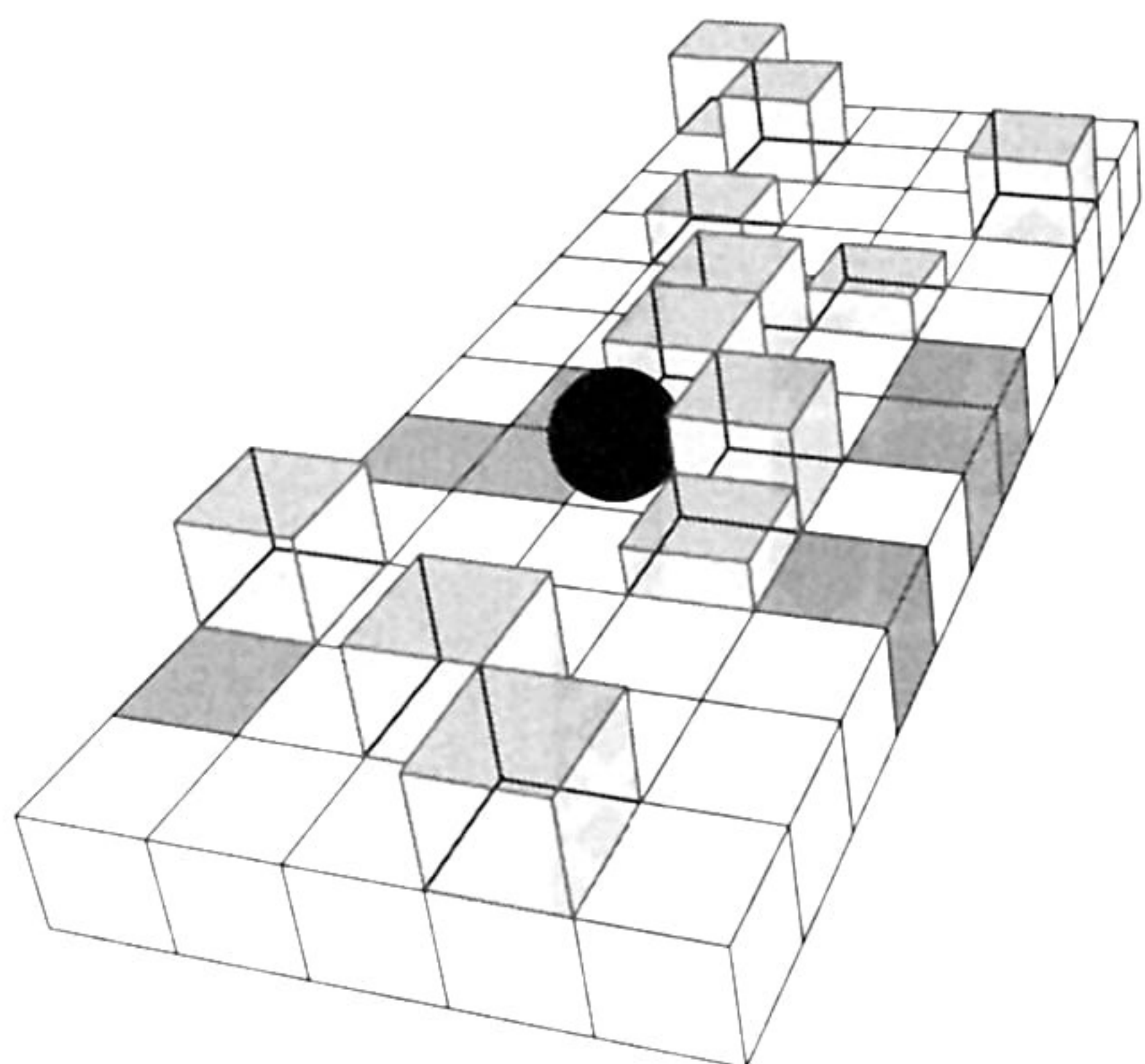
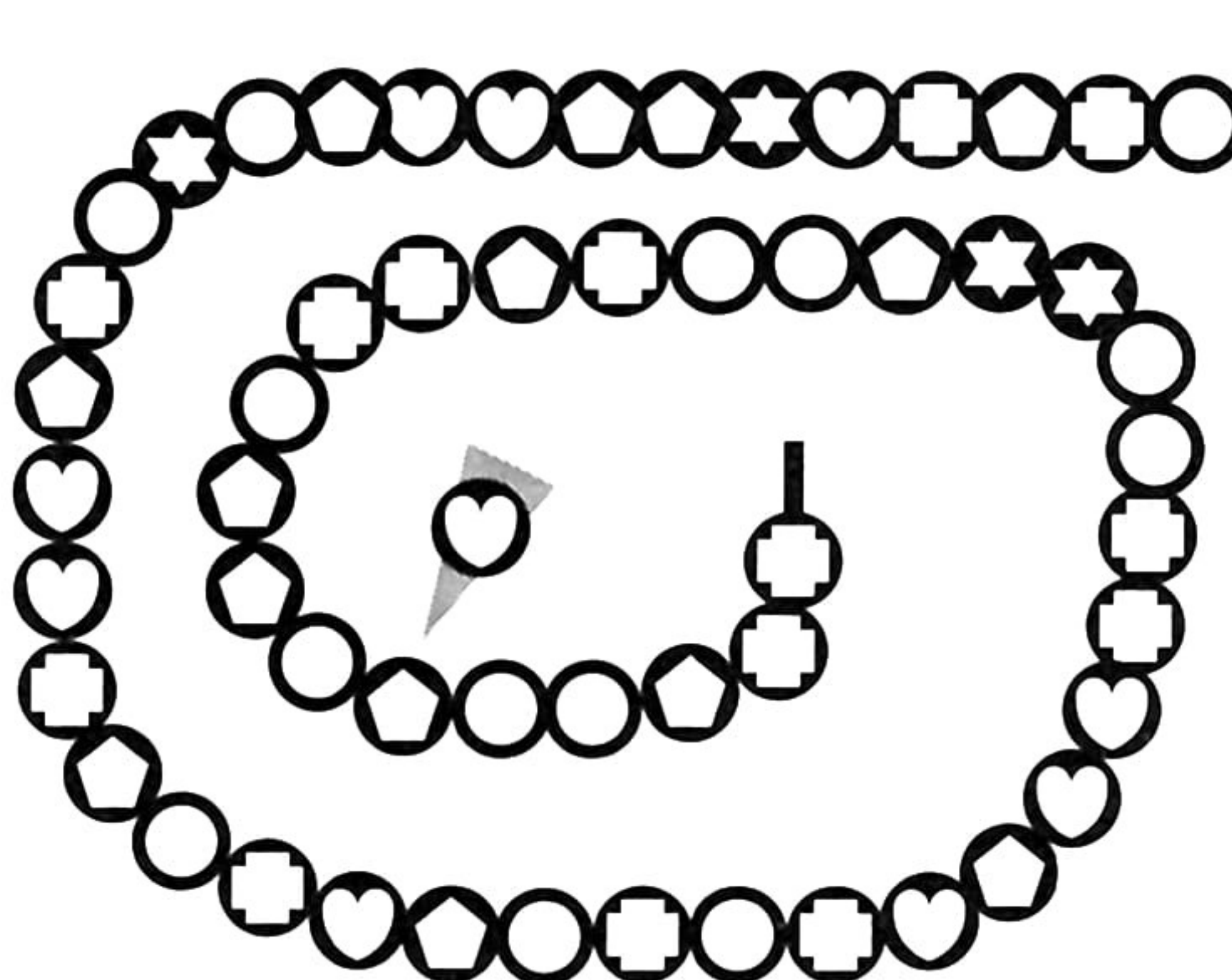



Fig. 0-6 軌道に沿って進むボールのゲーム

パズルゲームを作るには

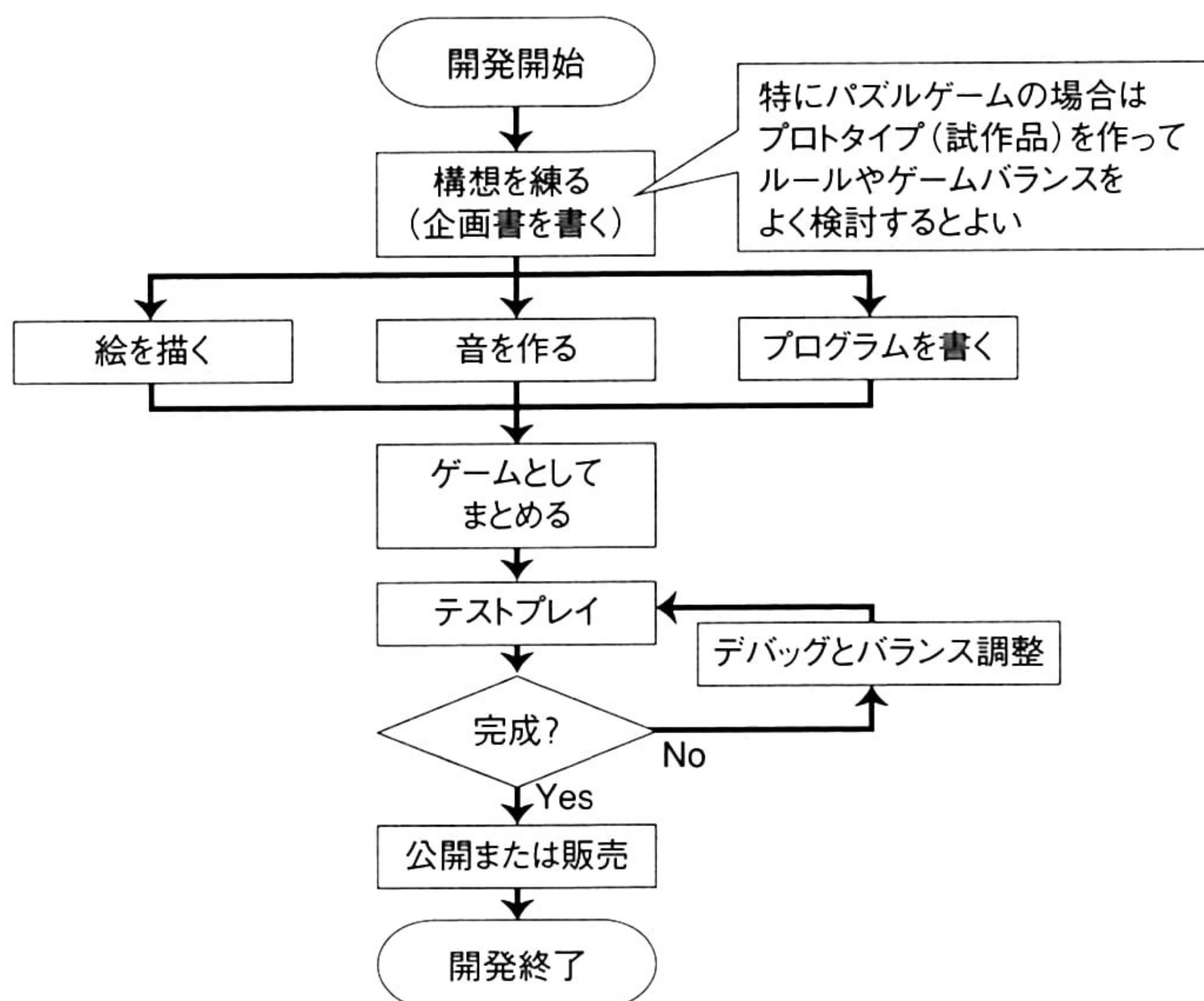
パズルゲームを作る手順は、趣味で作る場合も、仕事で作る場合も、あるいは学校の課題で作る場合も、基本はそれほど変わりません（Fig. 0-7）。開発に複数の人間が関わる場合には、いくつかの作業を並行して進めます。

開発の最初はゲームの構想を練ることから始まります。仕事の場合には企画書や仕様書を書いたり、開発予算を確保したりする必要もあります。基本的な構想がある程度できたら、絵・音・プログラムの作成に入ります。データやプログラムを作りながら、さらに構想をふくらませていくこともあるでしょう。

よいゲームを作るために大事なのはテストプレイを重ねることです。単にゲームとして動けばよいというだけではなく、時間の許すかぎりテストプレイ・デバッグ・ゲームバランス調整を繰り返します。

特にパズルゲームの場合には、グラフィックやサウンドもさることながら、ルールやゲームバランスがとても重要です。最初から凝った絵や音を用意するのではなく、絵や音は簡単な仮のものでもかまわないので、まずゲームのプロトタイプ（試作品）を作ってみるとよいでしょう。そして、面白いゲームができそうだという確信が持てるまで、プロトタイプを遊びこんで、検討を重ねるべきです。

Fig. 0-7　パズルゲームを作る手順

開発開始
構想を練る（企画書を書く）
特にパズルゲームの場合はプロトタイプ（試作品）を作ってルールやゲームバランスをよく検討するとよい
絵を描く
音を作る
プログラムを書く
ゲームとしてまとめる
テストプレイ
デバッグとバランス調整
完成？
No
Yes
公開または販売
開発終了

サンプルプログラムの紹介

本書で紹介するさまざまなアルゴリズムが実際に動いているところを確認できるように、ゲームふうのサンプルを用意し、付録CD-ROMに収録しました。実際にキャラクターやボールなどを操作して、荷物を押したり、ボールを積み上げげたり、ブロックを並べて消したりすることができます。

本書はパズルゲームに登場するさまざまなアルゴリズムを紹介することを主題としているため、ゲームの枠組みを作る方法については解説を省略しています。しかし、付録CD-ROMに収録したサンプルは、移動・描画・キー操作など、ゲームの基本的な処理を備えているので、ご安心ください。サンプルのソースコードが、ゲームの基本的な部分を作成する際の参考になってくれるでしょう。

また、本書内で掲載しているソースコードは、各アルゴリズムの中核となる部分を抜粋したものです。関連する処理やクラスの定義については、サンプル内の各アルゴリズムに対応する部分をご参照ください。

サンプルプログラムの実行方法

本書の付録CD-ROMには、サンプルプログラムのソースファイル・データファイル・実行ファイルの一式が収録されています。サンプルを実行するには、実行ファイル（Puzzle￥Release￥Puzzle.exeまたはPuzzle￥Debug￥Puzzle.exe）を、エクスプローラから実行してください。各サンプルの内容は、本書の対応する章で紹介しています。

サンプルプログラムはWindows Vista/XP/2000に対応しています。また、サンプルを実行する場合には、事前に下記のソフトウェアをインストールしておく必要があります。

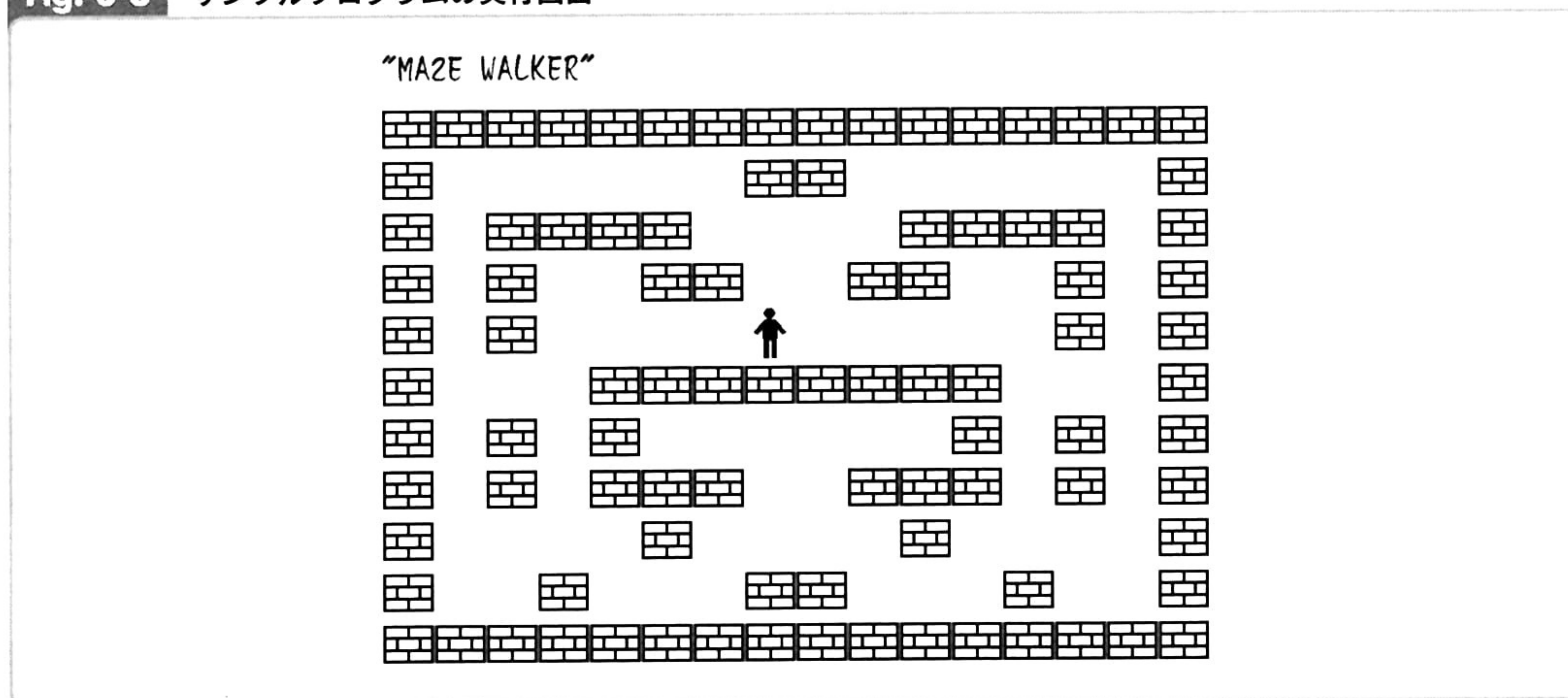
・Microsoft Visual C++ 2008 再頒布可能パッケージ（x86）
・DirectX エンドユーザー ランタイム（June 2008）

URL　http://www.microsoft.com/japan/downloads/

サンプルプログラムを起動すると、ゲーム画面が表示されます（Fig. 0-8）。
この画面では、ジョイスティックまたはキーボードで以下のような操作ができます。

・レバーの左右（カーソルキーの左右）でステージの選択（1ステージ単位）
・レバーの上下（カーソルキーの上下）でステージの選択（10ステージ単位）
・ボタン4（Bキー）でステージの開始
・ボタン5（Nキー）で一時停止
・Escキーでプログラムの終了

Fig. 0-8 サンプルプログラムの実行画面

"MAZE WALKER"

サンプルプログラムには、本書で紹介する数々のアルゴリズムに対応したステージが、掲載順に並んでいます。好きなステージを選択して、開始してみてください。ステージを開始した後は、以下のような操作ができます。

- レバーの上下左右（カーソルキーの上下左右）でキャラクターやカーソルの移動
- ボタン0～3（Zキー、Xキー、Cキー、Vキー）でアクションの実行（ステージによって内容は異なる）
- ボタン4（Bキー）でステージ選択に戻る
- ボタン5（Nキー）で一時停止
- Escキーでプログラムの終了

サンプルプログラムの起動時に、コマンドラインオプションに「-w 1024 -h 768」などを指定すると、解像度を変更することができます。フルスクリーン時の解像度は「-fw 1280 -fh 1024」のように指定します。

サンプルプログラムのビルド方法

本書では開発環境にVisual C++とDirectXを使います。付録CD-ROMに収録されたサンプルプログラムをビルドする場合には、事前に下記のソフトウェアをインストールしておく必要があります。サンプルプログラムを実行するだけならば、これらのソフトウェアは必要ありません。

- Visual C++ 2008 Express Edition

  URL　http://www.microsoft.com/japan/msdn/vstudio/express/

・DirectX SDK (March 2008)

URL ***http://www.microsoft.com/japan/msdn/directx/***

「Visual C++ 2008 Express Edition」はVisual C++ 2008の無償版です。本書では、開発環境をお持ちでない方にも、気軽にサンプルプログラムに触れていただくために、この製品を使うことにしました。詳しいインストール方法については、下記のページを参照してください。

・はじめての方のためのVisual C++ 2008 Express Editionインストール方法紹介

URL ***http://www.microsoft.com/japan/msdn/vstudio/express/beginners/2008/visualc.aspx***

・Visual Studio 2008 Express EditionのDVDイメージからのインストール

URL ***http://www.microsoft.com/japan/msdn/vstudio/express/maninstall/2008/***

サンプルプログラムをビルドするには、ソリューションファイル(Puzzle¥Puzzle.sln)をVisual C++ 2008 Express Editionで開きます。メインメニューの[ビルド]→[ソリューションのビルド]や[ビルド]→[ソリューションのリビルド]を選択すると、サンプルをビルドすることができます。本書の解説を参考に、パラメータなどを変更してビルドし、動きの変化などを確かめてみてください。

まとめ

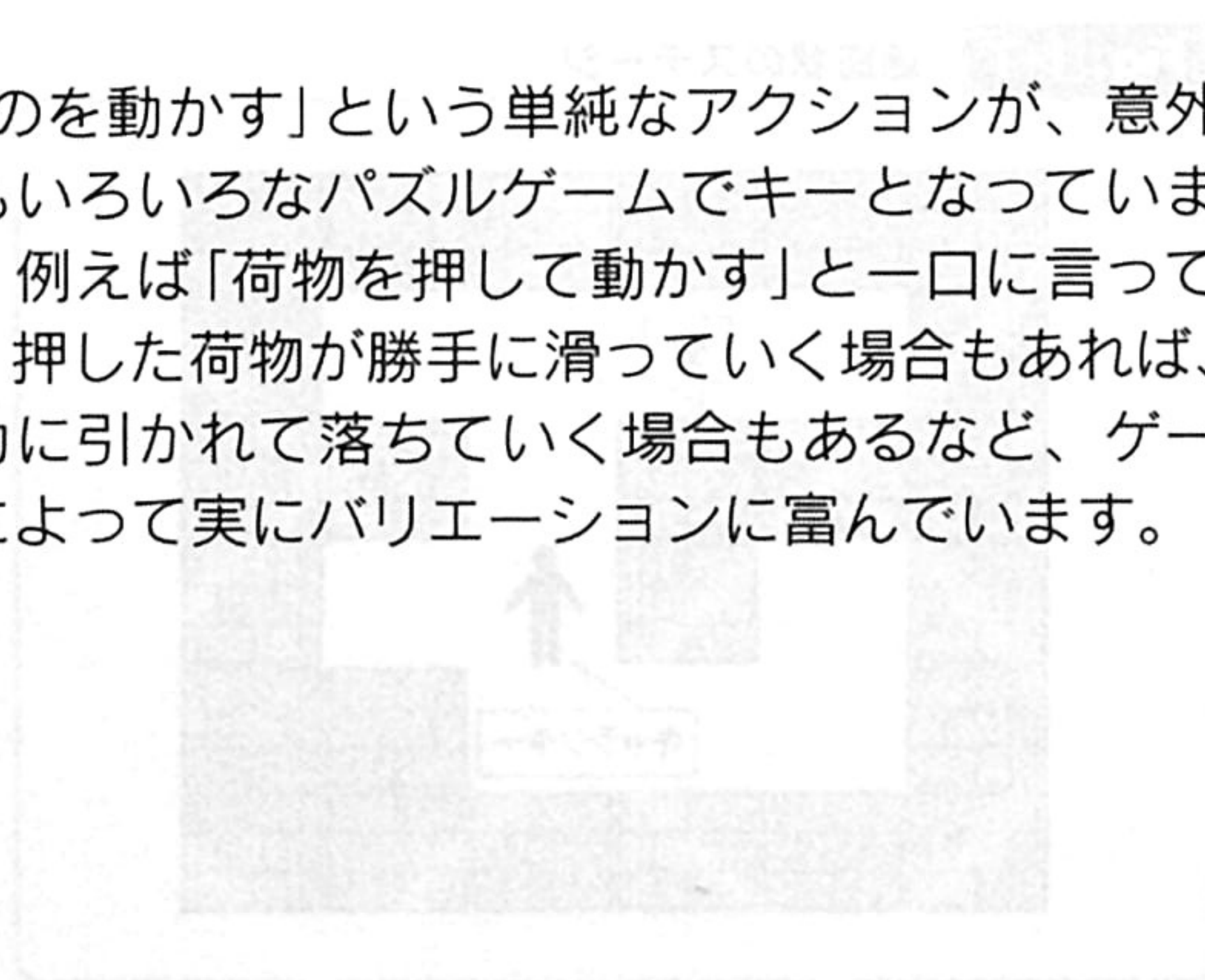
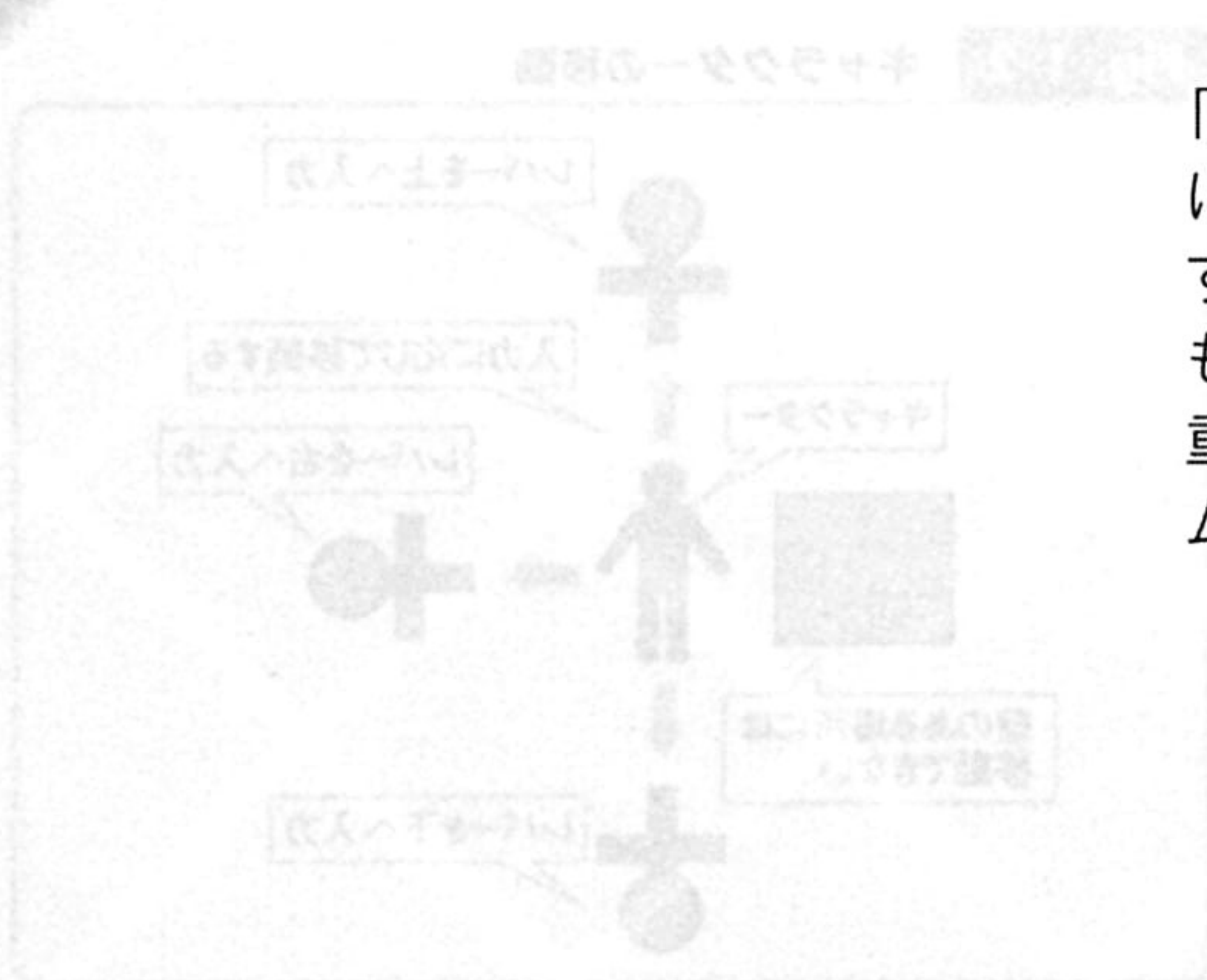
本章ではパズルゲームの全体像を概観し、プログラミングの要点について整理しました。ゲームプログラミングは決して難しくはありませんが、環境を準備したり、ライブラリの性格を覚えたりと、最初はいろいろと大変なことがあります。まずは、本書のサンプルを動かしたり、ビルドしたり、少し手を加えたりといったところから始めてみてください。きっと、楽しくスムーズにゲームプログラミングの世界に入っていけるでしょう。

というわけで、「パズルゲームを作るには、まずは気楽にゲームを遊んだり、手を加えたりするところから始めよう!」というのが本章のまとめです。

Stage 01

動かす

Move

「ものを動かす」という単純なアクションが、意外にもいろいろなパズルゲームでキーとなっています。例えば「荷物を押して動かす」と一口に言っても、押した荷物が勝手に滑っていく場合もあれば、重力に引かれて落ちていく場合もあるなど、ゲームによって実にバリエーションに富んでいます。

迷路を歩く

迷路状になったステージのなかで、キャラクターが歩き回るアクションです。プレイヤーのレバー入力やボタン入力に応じて、キャラクターは上下左右に動きます。

多くのゲームでは、壁などを用いて迷路を作ります（Fig. 1-1）。キャラクターはレバー入力に応じて上下左右に動くことができますが、壁のある場所に移動することはできません（Fig. 1-2）。

このように迷路のなかを歩くアクションは、非常に多くのゲームで採用されています。パズルゲームでの採用例としては、『倉庫番』や『フラッピー』などがあります。

ほとんどのゲームは、迷路のなかを歩くだけではなく、なんらかの特別な目的が用意されています。例えば『倉庫番』や『フラッピー』では、荷物やブロックを所定の位置に運ぶことがゲームの目的です。

アルゴリズム

迷路を使ったゲームでは、多くの場合、壁などを格子状に並べてステージを構成します（Fig. 1-3）。本書では、このような格子中のマス1つ1つを「セル」と呼ぶことにします。

セルには「壁」や「空間」、あるいは「キャラクター」や「荷物」といった種類があります。さまざまな種類のセルを格子状に配置することによって、ステージを表現します。

セル単位ではなく、画面上の座標を用いるなどして、より細かな単位でステージを管理することも可能です。本書にも、「結合して形を作る」（→p. 166）や「軌道に沿って進むボール」（→p. 278）など、セルではなく画面上の座標を用いてキャラクターの場所を管理するサンプルが

Fig. 1-1 迷路状のステージ

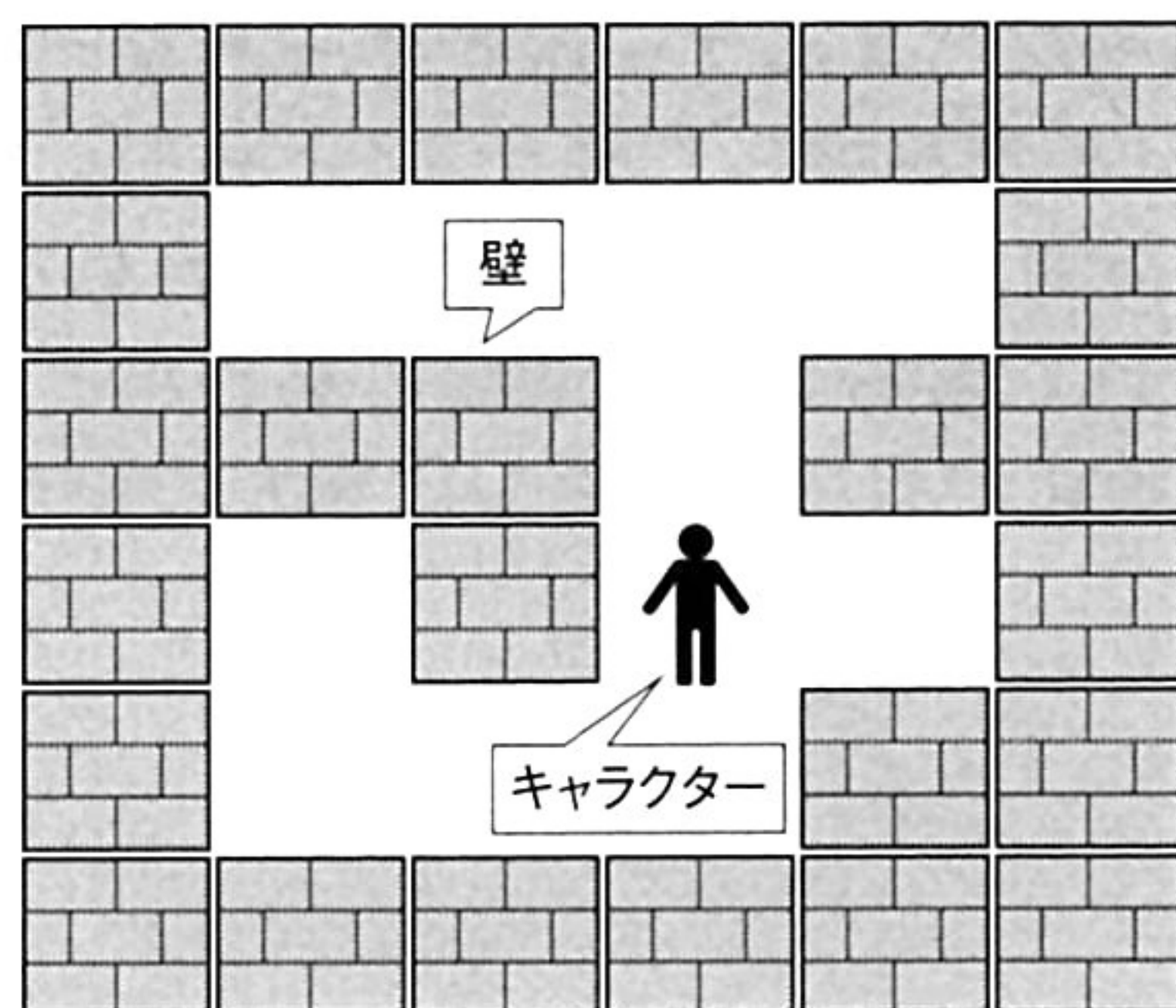



Fig. 1-2 キャラクターの移動

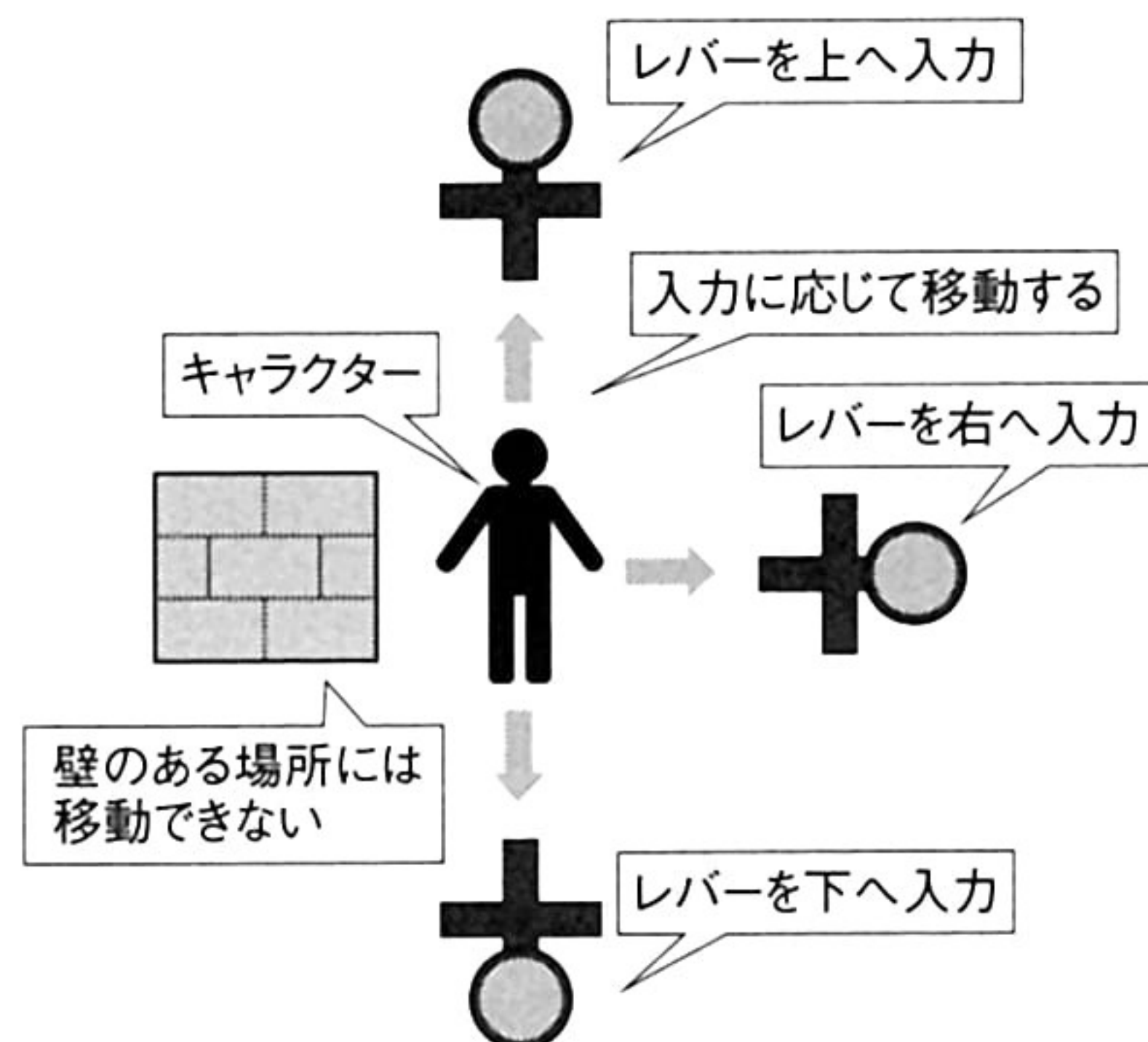
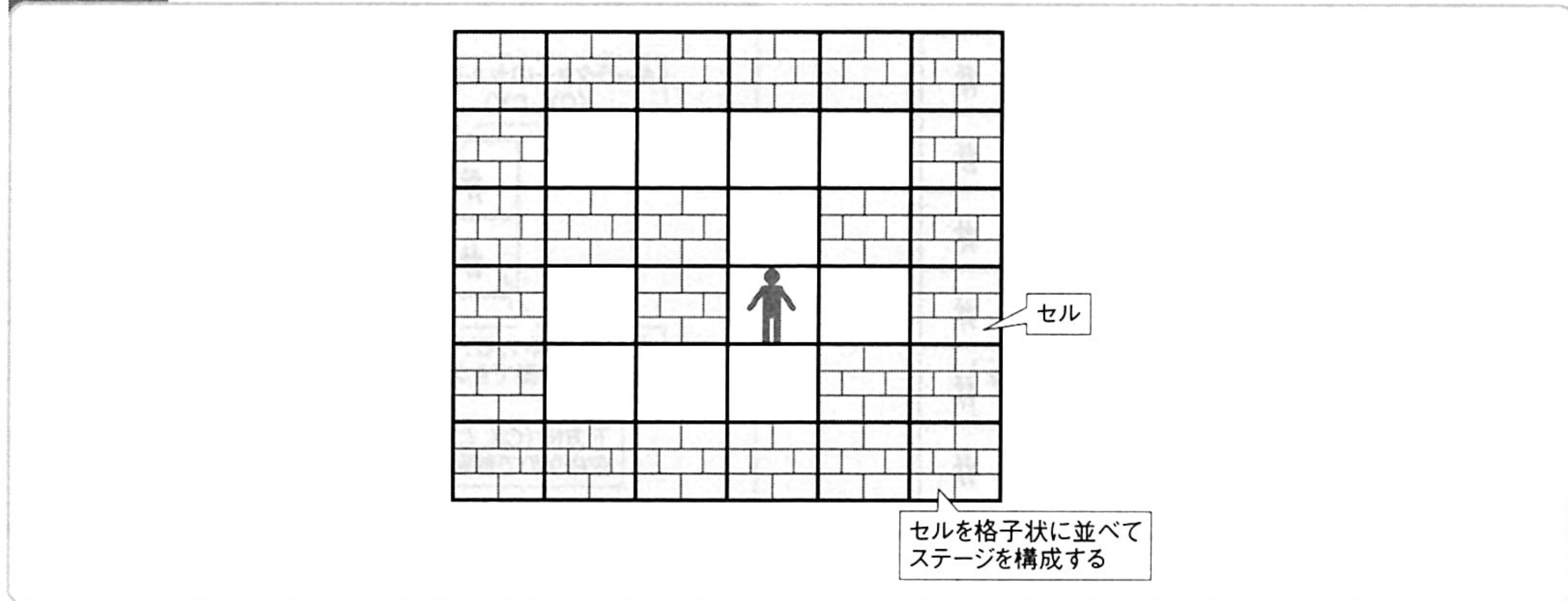



Fig. 1-3　セルを並べてステージを構成する

登場します。しかし、パズルゲームのなかには、移動をセル単位に限定した方が遊びやすくなるものが多くあります。そうすることで、極度に繊細な操作を要求されずに、パズルの解法に集中できるからです。

セルの管理

プログラムの内部では、セルの種類を文字や数値で表現すると便利です。例えば、壁を「#」で表し、キャラクターを「M」で表すと、ステージ全体はFig. 1-4のように表現できます。空白の部分は壁がないことを表しています。

通常、ゲームのステージは多数のセルから構成されています。そのため、プログラムは、すべてのセルに関する情報を記録しておく必要があります。後述するサンプルでは、配列変数を使ってセルの情報を管理しています。

なお、セルの種類と文字や数値の対応付けは、プログラムの都合に応じて自由に決めてかまいません。ここではなんとなくレンガの形に近い「#」で壁を表し、Man（人）の頭文字の「M」でキャラクターを表しましたが、もちろん他の文字を使うこともできます。

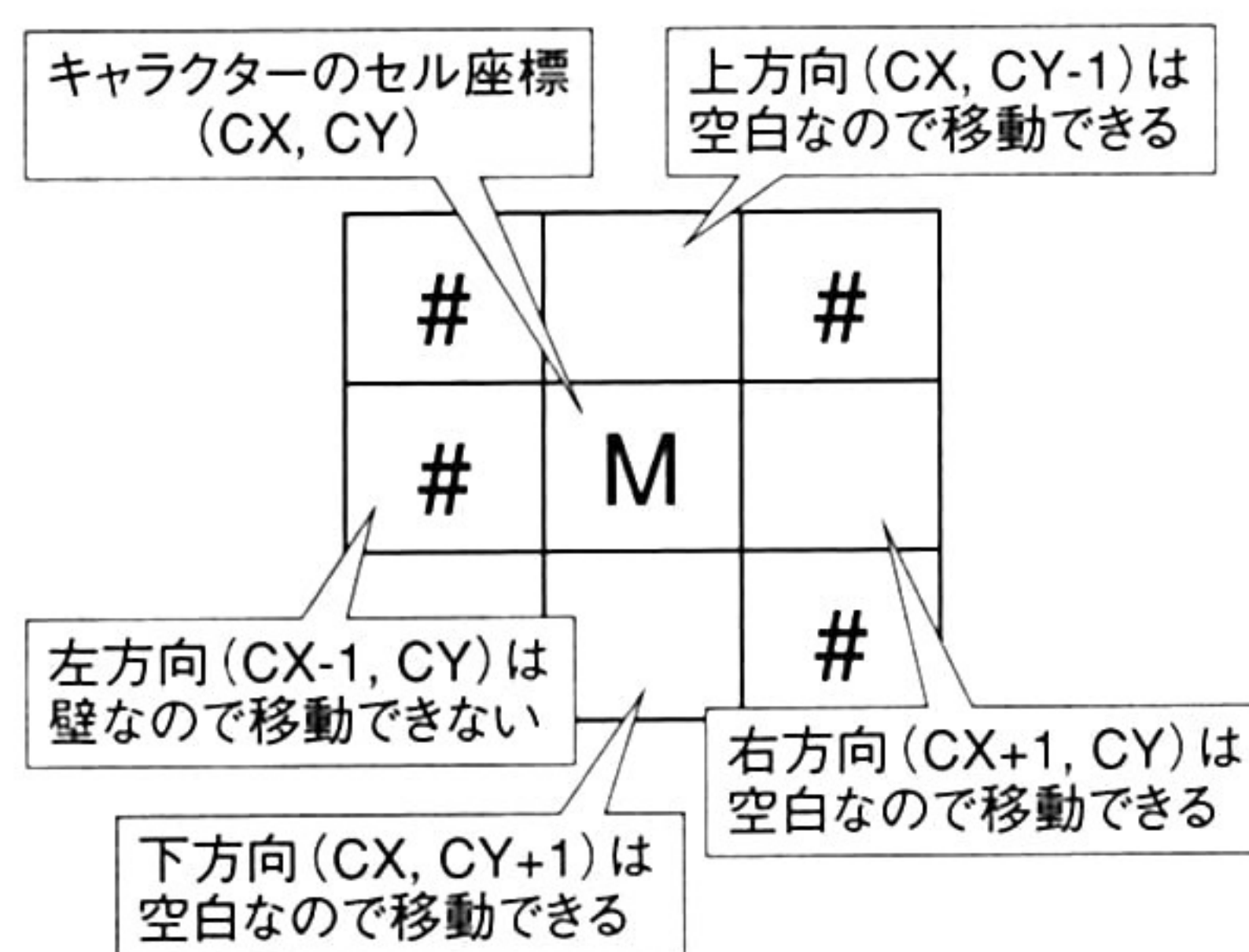
また、種類以外にもセルに関するさまざまな情報を管理したいときがあります。その場合は、文字や数値のかわりに、構造体やクラスなどを使うとよいでしょう。

キャラクターを歩かせるときには、セルを使って移動の可否を判定します（Fig. 1-5）。ここではセル単位の座標を「セル座標」と呼ぶことにしましょう。

キャラクターのセル座標を（CX, CY）とします。このとき、キャラクターの上下左右のセル座標は以下のとおりです。

上方向：（CX, CY-1）
下方向：（CX, CY+1）
左方向：（CX-1, CY）
右方向：（CX+1, CY）

Fig. 1-4 セルの種類を文字で表現する例

Fig. 1-5 セルを使った移動の判定

例えば、プレイヤーがレバーを上方向に入力したときには、セル座標（CX, CY-1）にある上方向のセルを調べます。セルが空白ならば、キャラクターは上に移動することができます。

また、レバーを左方向に入力したときには、セル座標（CX-1, CY）にある左方向のセルを調べます。左の方向が壁のセルならば、キャラクターは左に移動することができません。

キャラクターを移動させる

実際にキャラクターを移動させるためには、キャラクターのセル座標を更新します。例えばセル座標（CX, CY）のキャラクターを右に移動させるときには、CXに「1」を加算して、座標を（CX+1, CY）にします。

キャラクターの移動に合わせて、セルの内容も更新します（Fig. 1-6）。古い位置には空白を書き込み、新しい位置にはキャラクター（文字M）を書き込みます。

ゲームの内容にもよりますが、このようにセルを更新することは、キャラクターとさまざまな物体の干渉を判定する際に役立ちます。例えば「重力で落ちる荷物を押す」（→p. 30）では、荷物がキャラクターの頭上に載っているどうかを判定するために、セルを利用しています。

キャラクターのセル座標を更新したら、画面上のキャラクターも移動させます。ここで、画面にキャラクターを描画する際の座標を「描画座標」と呼ぶことにしましょう。

セル座標に合わせて描画座標も更新すれば、画面上でキャラクターが動きます（Fig. 1-7）。ここでは移動前の描画座標を（X, Y）とし、移動後の描画座標を（X+1, Y）としました。ただし、描画座標を一度に変化させてしまうと、キャラクターが滑らかに動かず、カクカクと移動してしまいます。

キャラクターを滑らかに移動させるためには、描画座標を少しずつ変化させます（Fig. 1-8）。移動前の描画座標を（X, Y）としたら、例えば、

（X+0.1, Y）
（X+0.2, Y）
（X+0.3, Y）
⋮

Fig. 1-6　移動にともなうセルの更新

#		#
#		M
		#

新しい位置のセルにキャラクター(文字M)を書き込む

古い位置のセルに空白を書き込む

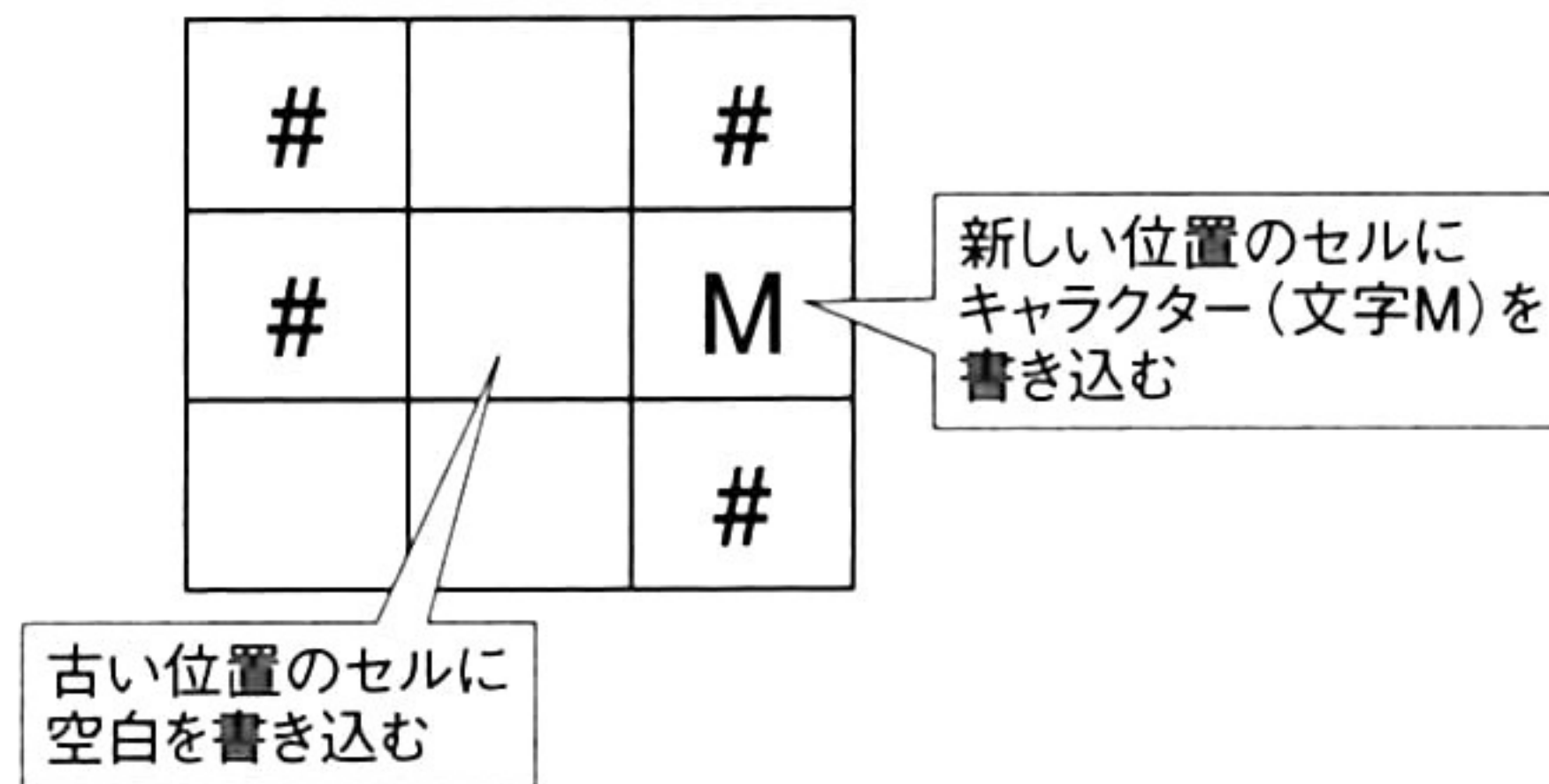



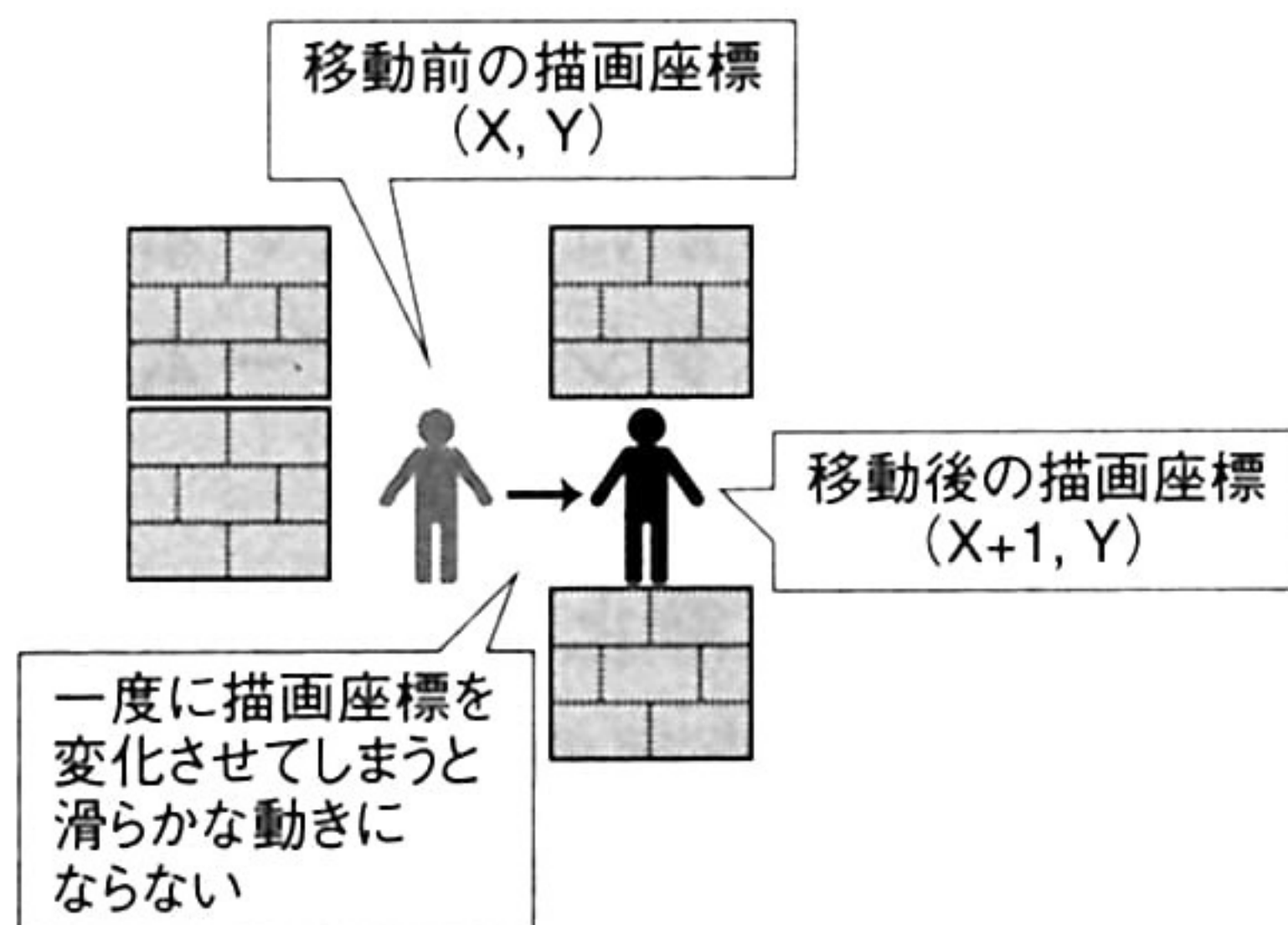
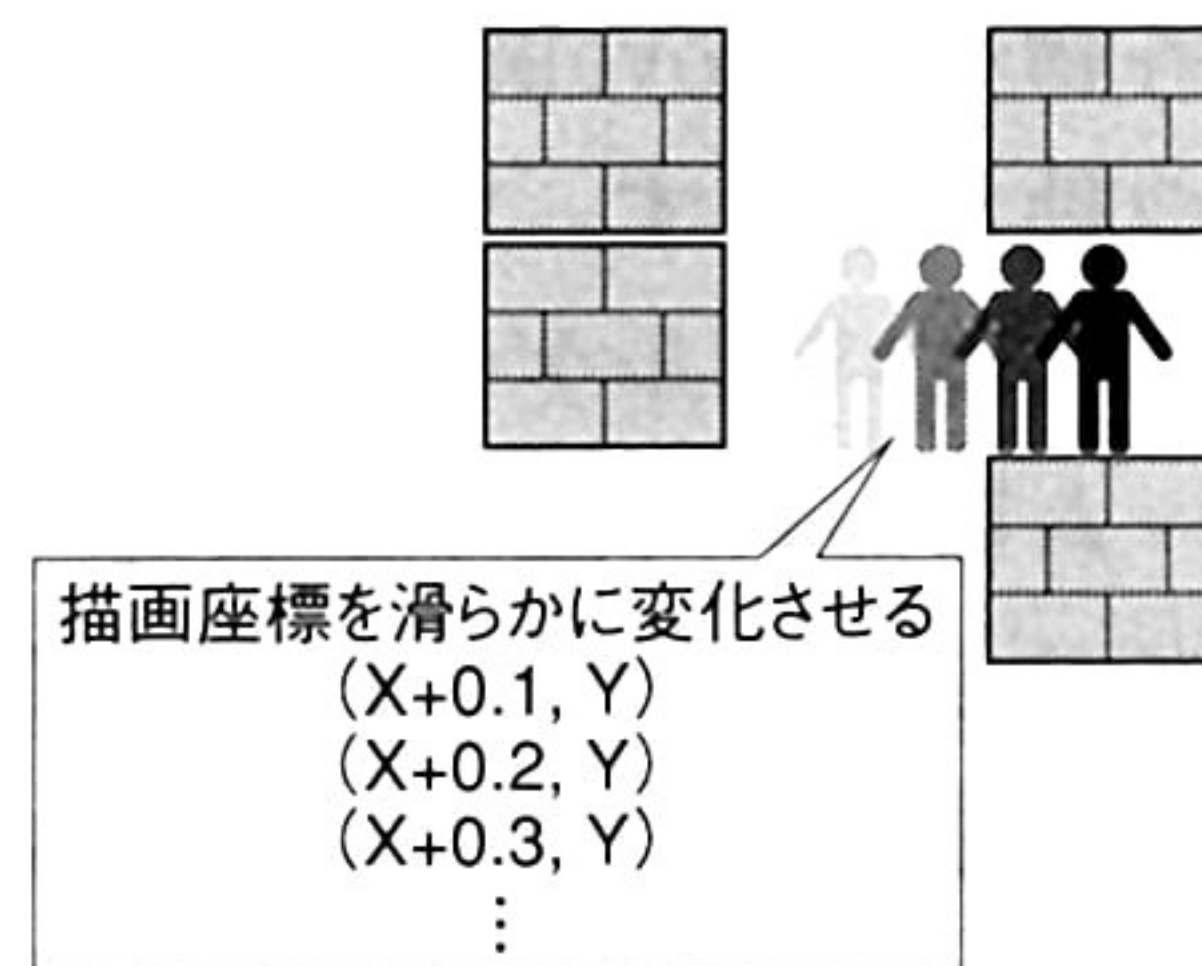
Fig. 1-7　滑らかではないキャラクターの移動

移動前の描画座標(X, Y)

移動後の描画座標(X+1, Y)

一度に描画座標を変化させてしまうと滑らかな動きにならない

Fig. 1-8　滑らかなキャラクターの移動

描画座標を滑らかに変化させる
(X+0.1, Y)
(X+0.2, Y)
(X+0.3, Y)
⋮

といった要領で、ある程度の時間を費やして、少しずつ描画座標を目標値に近づけます。これで画面上ではキャラクターが滑らかに動くようになります。

プログラム

List 1-1は迷路を歩くプログラムです。迷路とキャラクターを表示します。キャラクターはレバー入力(ジョイスティックの方向キー、またはキーボードのカーソルキー)で上下左右に移動することができます。ただし、壁のある場所には移動できません。

ここでは本書における最初の項目ということで、迷路を歩く処理に関するプログラムについて、少し長めに抜粋して掲載することにしました。他の項目では、プログラムから特に重要な処理だけを選び出して、できるだけコンパクトに掲載しています。

基本的な処理の流れやクラスの構成などは、どの項目でも共通です。他の項目を読むときに、プログラムの動きが把握しにくい場合には、本項目のプログラムと見比べてみてください。

クラスの構成

List 1-1は3つのクラスから構成されています。

セルを管理するクラス

最初のCCellクラスは、セルを管理するためのクラスです。格子状に並べられた多数のセルを、配列変数を使って管理します。また、指定した座標のセルを取得したり、設定したり、入れ替えたりといった処理を簡単に行うためのメソッドを提供します。

キャラクターのクラス

CMazeWalkerManクラスは、キャラクターのクラスです。プレイヤーのレバー入力を読み取って、キャラクターを上下左右に動かします。

このクラスのポイントはMove関数です。本書のサンプルでは、キャラクターやブロックやボールといった物体を動かす処理を「移動処理」と呼び、Move関数にまとめて記述しています。そして、各物体のMove関数を一定時間（1/60秒）ごとに繰り返し呼び出すことによって、物体を少しずつ動かします。わずかな動きを繰り返すことで、物体が滑らかに動いているように見えます。これはパズルゲームにかぎらず、シューティングゲームやアクションゲームなど、あらゆるゲームにおいて基本的な処理です。

CMazeWalkerManクラスの移動処理では、キャラクターの状態を「静止状態」と「移動状態」に分類しています。このサンプルでは、静止状態はキャラクターが特定のセルにいる状態で、移動状態はキャラクターがセル間を動いている途中の状態です。なお、状態の名称や内容はサンプルによって異なります。静止状態ではレバー入力に応じて移動方向を決め、移動状態に移行します。移動状態ではレバー入力は受け付けずに、決められた移動方向へ滑らかに移動します。隣接するセルへの移動が完了すると、静止状態に戻ります。

このようにキャラクターの動きをいくつかの状態に分類するのは、プログラミングをしやすくするためです。実際のゲームには静止状態や移動状態だけではなく、無敵状態や爆発状態など、さまざまな状態があります。

ステージのクラス

最後のCMazeWalkerStageクラスは、ステージのクラスです。サンプルには数々のステージが収録されており、各ステージが本書の各項目に対応しています。

このクラスのポイントはInit関数とDraw関数です。Init関数はステージの初期化を行います。セルの内容を初期化したり、キャラクターを配置したりします。一方、Draw関数はステージを描画します。このサンプルでは、ステージ内のセルを調べて、壁のセルに相当する位置に壁のグラフィックを表示します。

List 1-1 迷路を歩く（CCellクラス、CMazeWalkerManクラス、CMazeWalkerStageクラス）

```
// セルを管理するクラス
class CCell {

    // セルの種類を表す配列
    char* Cell;
```

```
    // X方向（横方向）のセル数、Y方向（縦方向）のセル数
    int XSize, YSize;

public:

    // コンストラクタ（セル数を指定する）
    // セルの配列を確保する
    CCell(int xsize, int ysize)
    :   XSize(xsize), YSize(ysize)
    {
        Cell=new char[XSize*YSize];
    }

    // コンストラクタ（デフォルトのセル数を使う）
    // セルの配列を確保する
    CCell()
    :   XSize(MAX_X), YSize(MAX_Y)
    {
        Cell=new char[XSize*YSize];
    }

    // デストラクタ
    // セルの配列を削除する
    ~CCell() {
        delete[] Cell;
    }

    // セルの初期化
    // 引数に指定された内容でセルの配列を初期化する
    void Init(char* cell) {
        for (int i=0, n=XSize*YSize; i<n; i++) Cell[i]=cell[i];
    }

    // セルの取得
    // 指定された座標にあるセルの種類を取得する
    char Get(int x, int y) {
        if (0<=x && x<XSize && 0<=y && y<YSize) {
            return Cell[x+y*XSize];
        } else {
            return ' ';
        }
    }

    // セルの設定
    // 指定された座標にあるセルの種類を設定する
    void Set(int x, int y, char value) {
        if (0<=x && x<XSize && 0<=y && y<YSize) {
            Cell[x+y*XSize]=value;
        }
    }
```

```
    // セルの入れ替え
    // 指定された2つの座標にあるセルを入れ替える
    void Swap(int xa, int ya, int xb, int yb) {
        char c=Get(xa, ya);
        char d=Get(xb, yb);
        Set(xa, ya, d);
        Set(xb, yb, c);
    }
};

// キャラクターのクラス
class CMazeWalkerMan : public CMover {
public:

    // ステージを表すセル
    CCell* Cell;

    // セル上の座標、移動方向
    int CX, CY, VX, VY;

    // キャラクターの状態、タイマー
    int State, Time;

    // コンストラクタ
    // 座標、状態、タイマー、画像などを設定する
    CMazeWalkerMan(CCell* cell, int x, int y)
    :    Cell(cell), CX(x), CY(y), VX(0), VY(0), State(0), Time(0)
    {
        Texture=Game->Texture[TEX_MAN];
        X=x;
        Y=y;
    }

    // 移動
    // レバー入力に応じてキャラクターを上下左右に動かす
    // この処理は一定時間(1/60秒)ごとに呼び出される
    virtual bool Move(const CInputState* is) {

        // 静止状態(キャラクターが静止している状態)
        if (State==0) {

            // レバー入力に応じて移動方向を設定する
            VX=VY=0;
            if (is->Left) VX=-1; else
            if (is->Right) VX=1; else
            if (is->Up) VY=-1; else
            if (is->Down) VY=1;
```

```
            // レバーが入力された場合の処理
            if (VX!=0 || VY!=0) {

                // 移動先にあるセルが空（空白文字）かどうかを調べる
                if (Cell->Get(CX+VX, CY+VY)==' ') {

                    // 移動先が空の場合にはキャラクターを移動させる
                    // 現在キャラクターがいるセルを空にして、
                    // 移動先のセルにキャラクター（文字M）を設定する
                    Cell->Set(CX, CY, ' ');
                    Cell->Set(CX+VX, CY+VY, 'M');

                    // セル上のキャラクター座標を更新する
                    CX+=VX;
                    CY+=VY;

                    // タイマーを設定して、移動状態に移行する
                    State=1;
                    Time=10;
                }
            }
        }

        // 移動状態（キャラクターが移動中の状態）
        if (State==1) {

            // タイマーを減少させる
            // タイマーを増加させる処理にしてもよい（Stage02以降のプログラムを参照）
            Time--;

            // タイマーを使って、画面上のキャラクター座標を少しずつ変化させることにより、
            // キャラクターを滑らかに移動させる
            X=CX-VX*Time*0.1f;
            Y=CY-VY*Time*0.1f;

            // タイマーが0になったら、移動方向を初期化し、静止状態に戻る
            if (Time==0) {
                VX=VY=0;
                State=0;
            }
        }

        // 本書のサンプルの移動処理では、
        // キャラクターを消去しない場合にはtrueを返し、
        // 消去する場合にはfalseを返すことにしている
        return true;
    }
};
```

```
// ステージのクラス
class CMazeWalkerStage : public CStage {

    // ステージを表すセル
    CCell* Cell;

public:

    // コンストラクタ
    // ステージ名を設定し、セルを表すオブジェクトを作成する
    CMazeWalkerStage() : CStage(L"MAZE WALKER") {
        Cell=new CCell();
    }

    // デストラクタ
    // セルのオブジェクトを廃棄する
    virtual ~CMazeWalkerStage() {
        delete Cell;
    }

    // ステージの初期化
    // この処理はステージを開始するたびに呼び出される
    virtual void Init() {

        // セルを初期化して迷路状のステージを作る
        Cell->Init(
            "                "
            "################"
            "#      ##      #"
            "# ####    #### #"
            "# #  ##  ##  # #"
            "# #    M     # #"
            "#   ########   #"
            "# # #      # # #"
            "# # ###  ### # #"
            "#    #    #    #"
            "#  #   ##   #  #"
            "################"
        );

        // すべてのセルを調べて、
        // キャラクターのセル(文字M)の位置に、キャラクターのオブジェクトを生成する
        for (int y=0; y<MAX_Y; y++) {
            for (int x=0; x<MAX_X; x++) {
                if (Cell->Get(x, y)=='M') {
                    new CMazeWalkerMan(Cell, x, y);
                }
            }
        }
    }
```

```
// ステージの描画
// この処理はステージを画面に描画する際に呼び出される
virtual void Draw() {

    // 画面の解像度に応じて描画サイズを変化させるための処理
    float
        sw=Game->GetGraphics()->GetWidth()/MAX_X,
        sh=Game->GetGraphics()->GetHeight()/MAX_Y;

    // すべてのセルを調べて、
    // 壁のセル(文字#)の位置に、壁の画像を描画する
    for (int y=0; y<MAX_Y; y++) {
        for (int x=0; x<MAX_X; x++) {
            if (Cell->Get(x, y)=='#') {
                Game->Texture[TEX_FLOOR]->Draw(
                    x*sw, y*sh, sw, sh, 0, 0, 1, 1, COL_BLACK
                );
            }
        }
    }
}
};
```

SAMPLE

「MAZE WALKER」は「迷路を歩く」のサンプルです。レバーの上下左右(カーソルキーの上下左右)でキャラクターが移動します。キャラクターが通れるのは通路だけで、壁を通過することはできません。

MAZE WALKER → p. 386

荷物を押す

キャラクターが荷物を押して動かすアクションです。荷物の種類はさまざまで、ゲームによっては岩やタル、フルーツなどを運ぶこともあります。

荷物を押すには、キャラクターを荷物に隣接させた後に、荷物を押すようにキャラクターを動かします(Fig. 1-9)。荷物の先に何も障害物がなければ、荷物はキャラクターに押されるように移動します。荷物の先に他の荷物がある場合には、荷物は動きません(Fig. 1-10)。荷物の先に壁がある場合にも、荷物は動きません(Fig. 1-11)。

荷物を押すアクションはいろいろなゲームに採用されています。よく知られているゲームとしては『倉庫番』があります。このゲームでは、ステージに配置された荷物を押して、決められた目的地に運びます。荷物を押すことはできますが、引くことはできないので、順番や経路

Fig. 1-9 荷物を押す

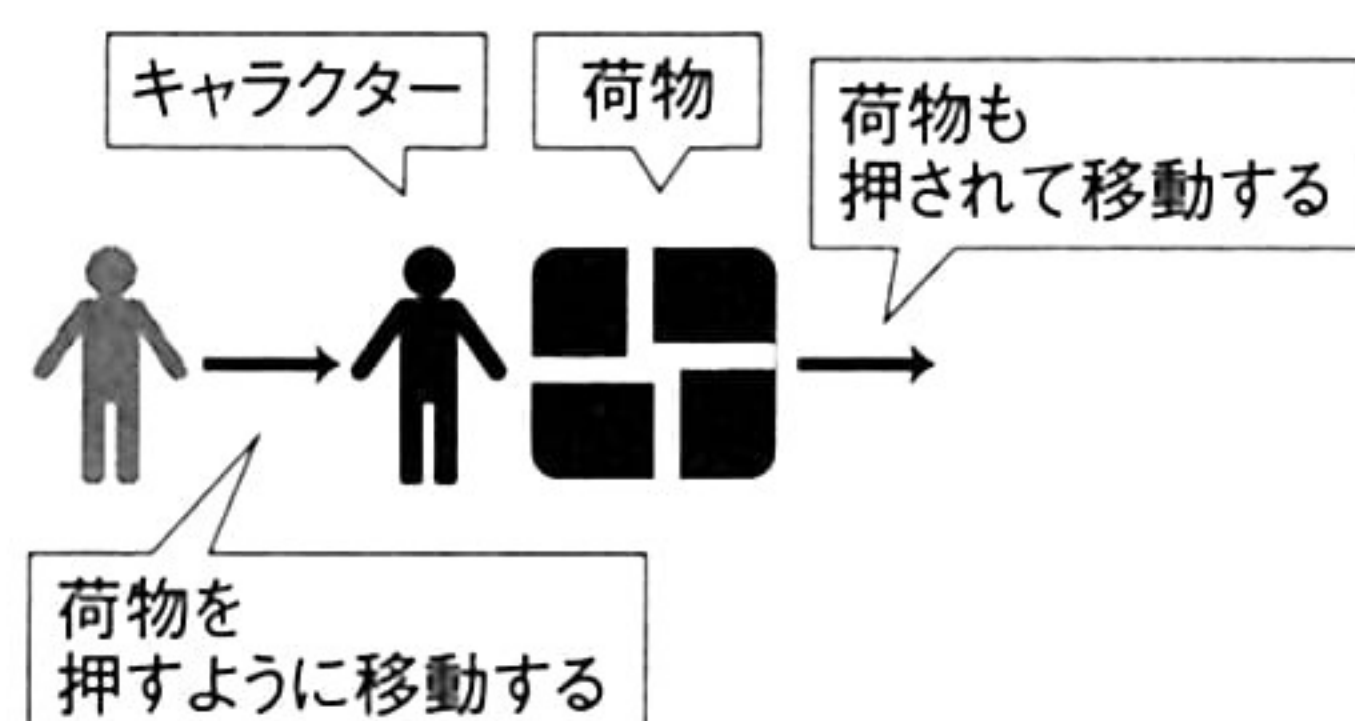
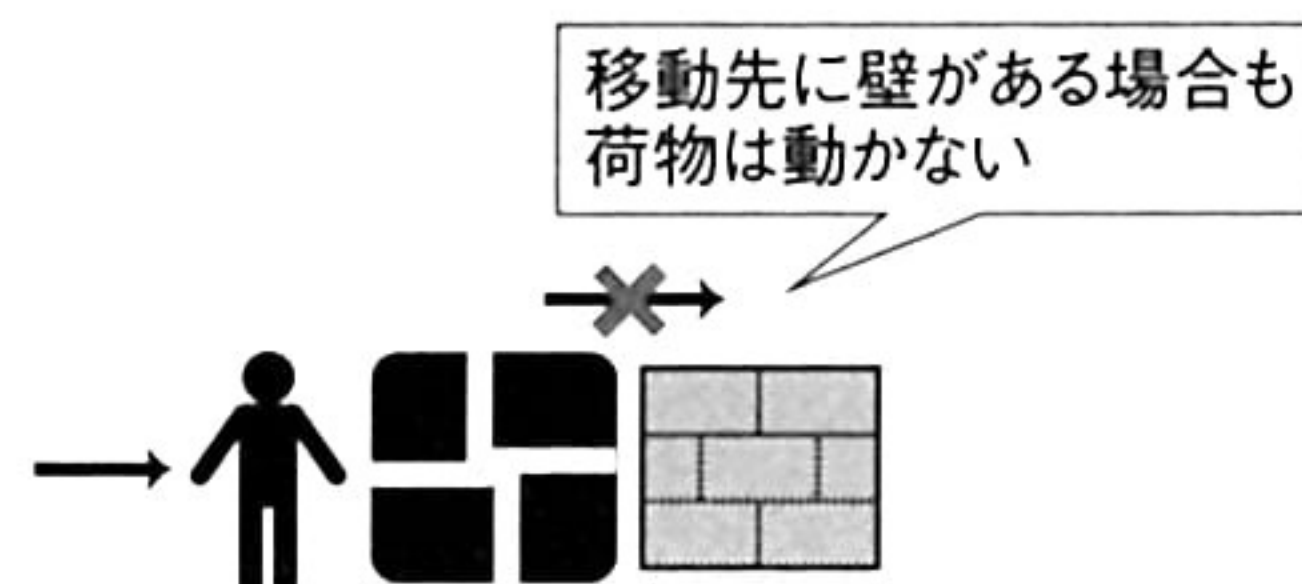



Fig. 1-10 荷物の先に他の荷物がある場合

Fig. 1-11 荷物の先に壁がある場合

を熟考して荷物を運ばないと、容易に行き詰まってしまいます。

こういったゲームが面白くなるかどうかは、魅力的なステージの配置を考え出せるかどうかにかかっています。もちろん難しいステージは魅力がありますが、難しいだけではなく、プレイヤーが面白みを感じるようなユニークな問題を考え出す必要があります。これは詰め将棋などの伝統的なパズルと同じです。ゲームを解くだけではなく、問題を作ることも、パズル好きにはたまらない作業でしょう。

アルゴリズム

荷物を押すアクションを実現するために、ここではステージをセルでとらえます。まずは、キャラクターが荷物を右方向に押そうとしている状況を考えましょう（Fig. 1-12）。

「迷路を歩く」（→p. 2）と同様にステージをセルで表現すると、Fig. 1-13のような状態になり

Fig. 1-12 荷物を押す

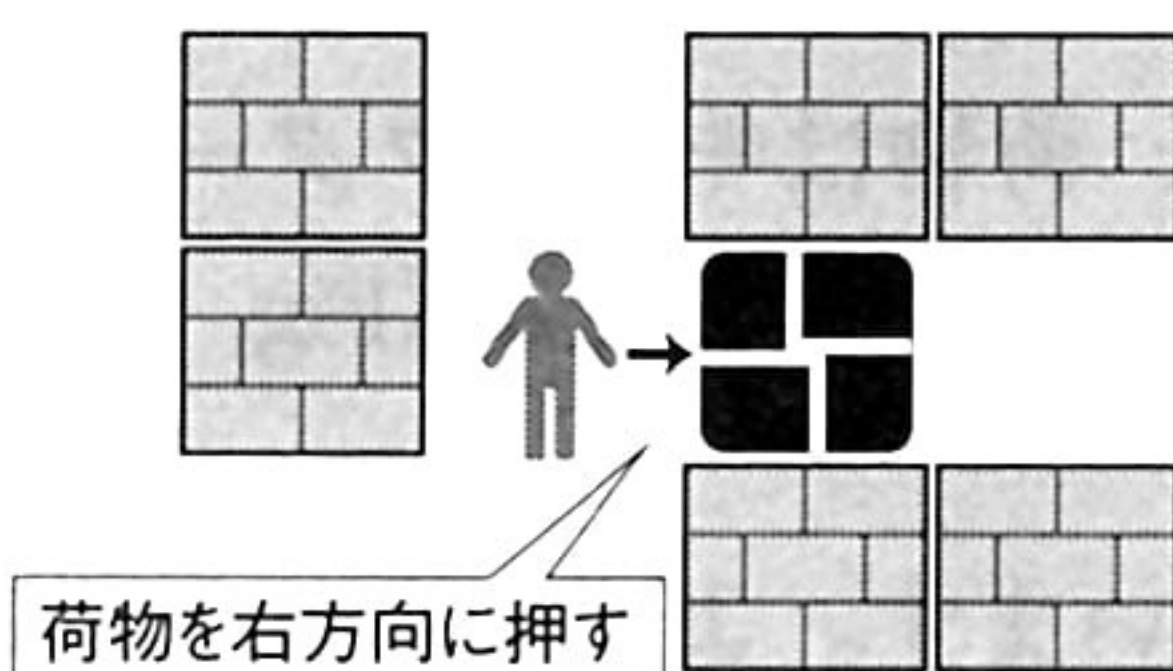



Fig. 1-13 荷物を押すときのセルの状態

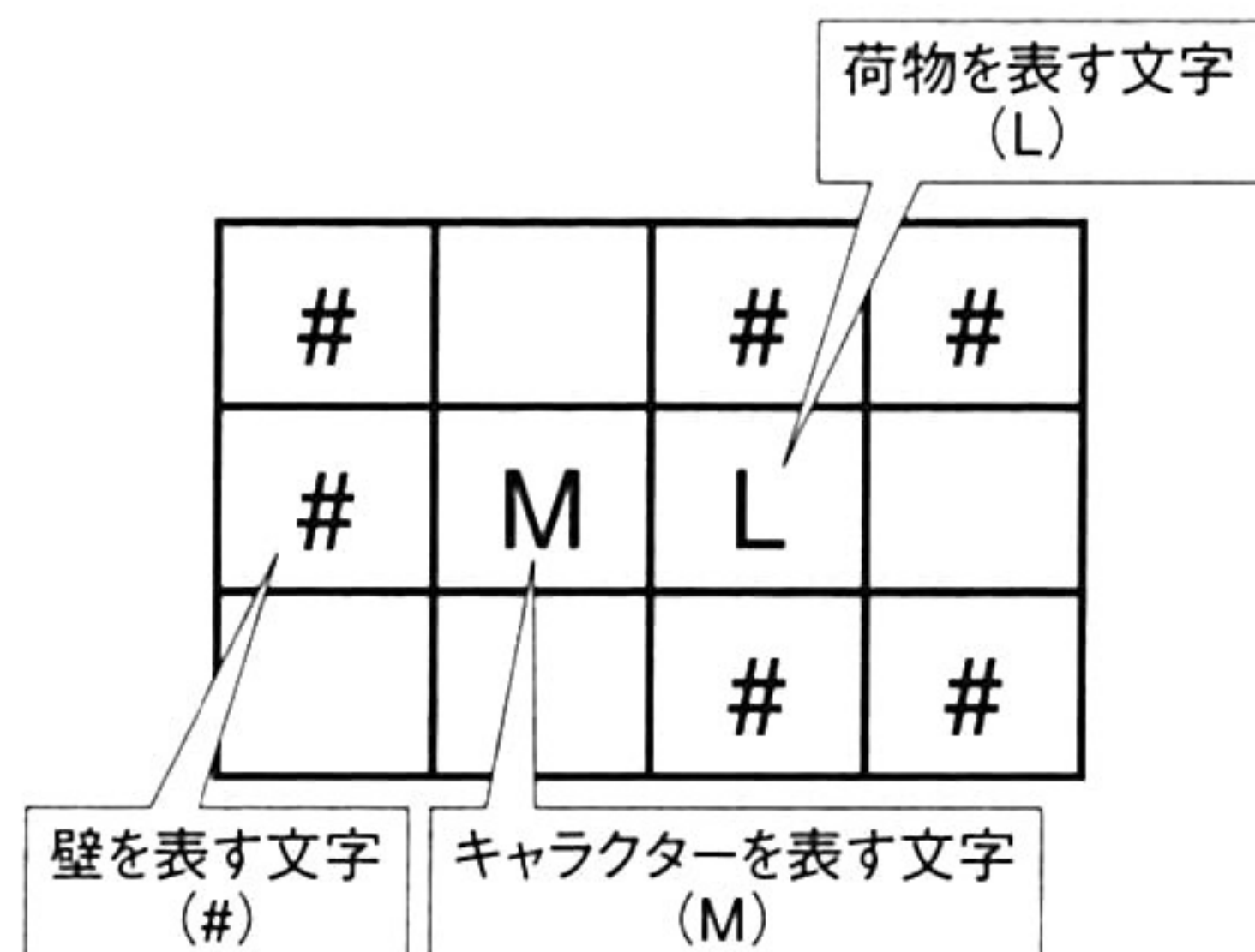
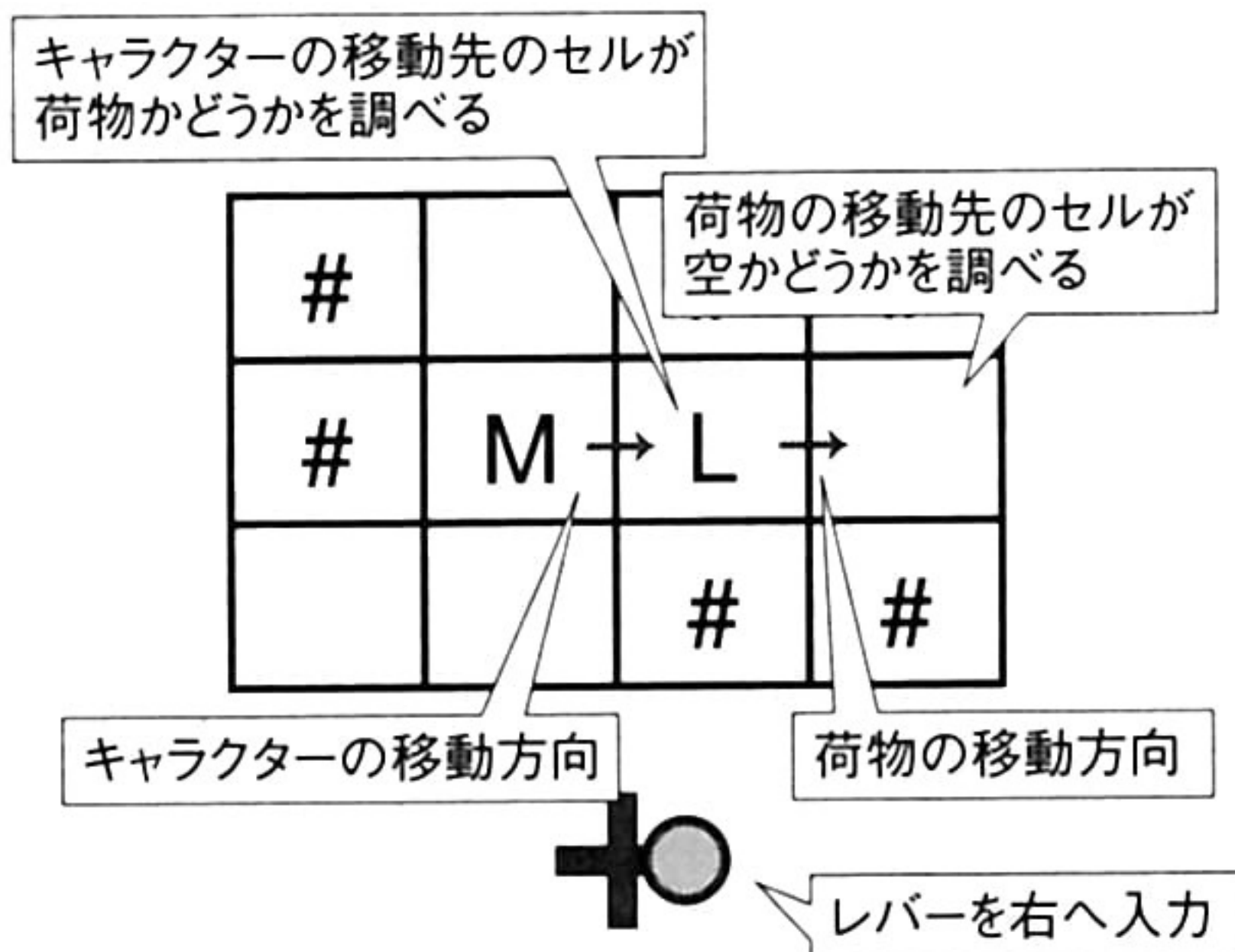
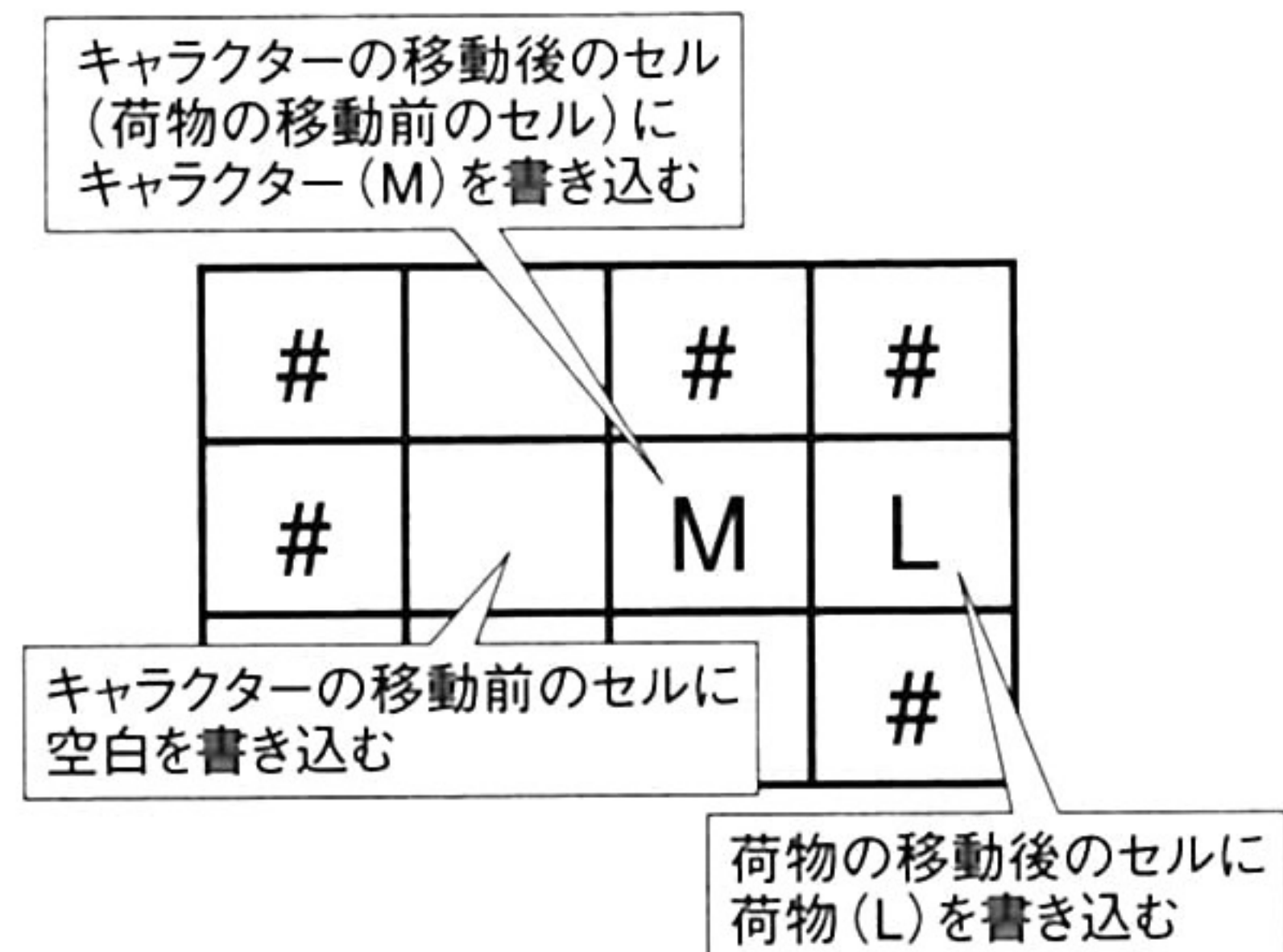
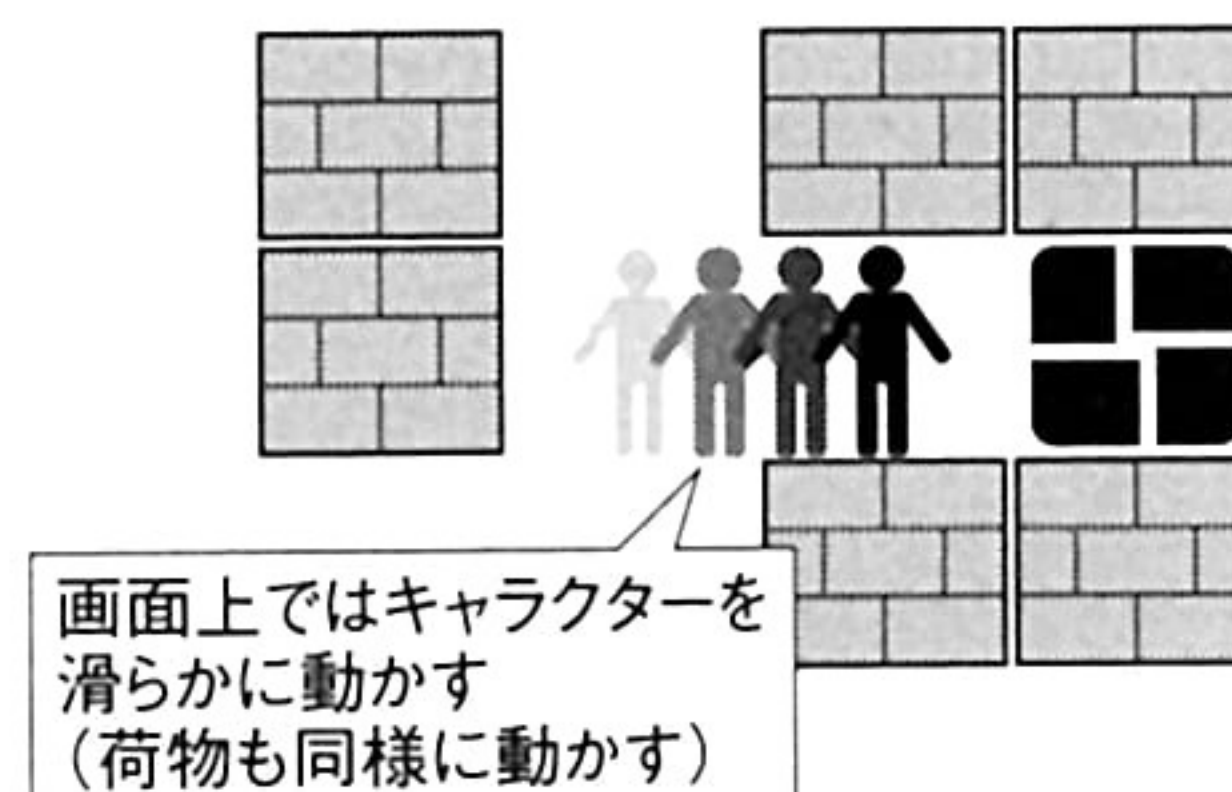
Fig. 1-14 移動先のセルを調べる

キャラクターの移動先のセルが荷物かどうかを調べる
荷物の移動先のセルが空かどうかを調べる
キャラクターの移動方向
荷物の移動方向
レバーを右へ入力

Fig. 1-15 セルの更新

キャラクターの移動後のセル（荷物の移動前のセル）にキャラクター（M）を書き込む
キャラクターの移動前のセルに空白を書き込む
荷物の移動後のセルに荷物（L）を書き込む

Fig. 1-16 画面の更新

画面上ではキャラクターを滑らかに動かす（荷物も同様に動かす）

ます。ここではキャラクターを文字「M」(Manの頭文字)、荷物を文字「L」(Loadの頭文字) で表現しました。

プレイヤーがレバーを入力したら、キャラクターの移動先のセルが荷物かどうかを調べます (Fig. 1-14)。もしも荷物の場合には、さらに荷物の移動先にあるセルが空かどうかを調べます。

荷物の移動先が空の場合には、荷物を押して動かすことができます。キャラクターと荷物について、それぞれ移動前と移動後のセルを更新して、移動後の状態を作り出します (Fig. 1-15)。

あとは新しいセルの状態に応じて画面を更新すれば完了です。「迷路を歩く」(→p. 10) と同様に、描画座標を少しずつ変化させることによって、キャラクターや荷物を滑らかに動かします (Fig. 1-16)。

プログラム

List 1-2は荷物を押すプログラムです。荷物とキャラクターの移動処理、および荷物を押すときの処理を掲載しました。

キャラクターと荷物には、それぞれ静止状態と移動状態があります。現在どちらの状態にあるのかは、変数Stateが示します。Stateの値によって、移動状態と静止状態を切り替えます。

キャラクターの場合、静止状態ではレバー入力に応じて移動方向を決定し、移動状態に移行します。移動状態では描画座標を少しずつ変化させることによって、画面上を滑らかに移動し

ます。

荷物の場合、キャラクターが荷物を押すと、静止状態から移動状態に移行します。移動状態では、キャラクターと同様に描画座標を少しずつ変化させて、画面上を滑らかに移動します。

キャラクターも荷物も、隣のセルへの移動が完了すると静止状態に戻ります。

List 1-2　荷物を押す(CLoadPusherLoadクラス、CLoadPusherManクラス)

```
//荷物を押すときに呼び出す処理
void CLoadPusherLoad::Push(int vx, int vy) {

    //荷物が静止状態ならば移動状態にする
    if (State==0) {

        //移動方向の設定
        VX=vx;
        VY=vy;

        //セル座標の更新
        CX+=vx;
        CY+=vy;

        //状態とタイマーの設定
        State=1;
        Time=10;
    }
}

//荷物の移動処理
bool CLoadPusherLoad::Move(const CInputState* is) {

    //荷物が移動状態のときの処理
    if (State==1) {

        //タイマーの更新
        Time--;

        //描画座標の更新
        //タイマーを使って少しずつ変化させる
        X=CX-VX*Time*0.1f;
        Y=CY-VY*Time*0.1f;

        //タイマーが0になったら静止状態に移行する
        if (Time==0) {
            State=0;
        }
    }
    return true;
}
```

```
//キャラクターの移動処理
bool CLoadPusherMan::Move(const CInputState* is) {

    //静止状態の処理
    if (State==0) {

        //レバー入力に応じて移動方向を設定する
        VX=VY=0;
        if (is->Left) VX=-1; else
        if (is->Right) VX=1; else
        if (is->Up) VY=-1; else
        if (is->Down) VY=1;

        //移動する場合の処理
        if (VX!=0 || VY!=0) {

            //移動先のセルが空の場合の処理
            if (Cell->Get(CX+VX, CY+VY)==' ') {

                //キャラクターの移動元と移動先のセルを更新する
                Cell->Set(CX, CY, ' ');
                Cell->Set(CX+VX, CY+VY, 'M');

                //セル座標の更新
                CX+=VX;
                CY+=VY;

                //状態とタイマーの設定
                State=1;
                Time=10;
            } else

            //移動先に荷物があり、
            //かつ荷物の移動先が空の場合の処理
            if (
                Cell->Get(CX+VX, CY+VY)=='L' &&
                Cell->Get(CX+VX*2, CY+VY*2)==' '
            ) {
                //キャラクターと荷物について
                //移動元と移動先のセルを更新する
                Cell->Set(CX, CY, ' ');
                Cell->Set(CX+VX, CY+VY, 'M');
                Cell->Set(CX+VX*2, CY+VY*2, 'L');

                //セル座標の更新
                CX+=VX;
                CY+=VY;

                //状態とタイマーの設定
```

```
                State=1;
                Time=10;

                //荷物のオブジェクトを探す
                for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
                    CMover* mover=(CMover*)i.Next();

                    //荷物を押すための処理を呼び出す
                    if ((int)mover->X==CX && (int)mover->Y==CY) {
                        ((CLoadPusherLoad*)mover)->Push(VX, VY);
                    }
                }
            }
        }
    }

    //移動状態の処理
    if (State==1) {

        //タイマーの更新
        Time--;

        //描画座標の更新
        //タイマーを使って少しずつ変化させる
        X=CX-VX*Time*0.1f;
        Y=CY-VY*Time*0.1f;

        //タイマーが0になったら静止状態に移行する
        if (Time==0) {
            State=0;
        }
    }
    return true;
}
```

SAMPLE

「LOAD PUSHER」は「荷物を押す」のサンプルです。レバーの上下左右(カーソルキーの上下左右)でキャラクターが移動します。

荷物に向かってキャラクターを動かすと、荷物を押して動かすことができます。荷物の先に別の荷物や壁があるときには、荷物を押すことはできません。

LOAD PUSHER → p. 386

滑る荷物を押す

キャラクターが押した荷物が、押した方向に滑っていくアクションです。荷物はいかにも滑りやすいもの、例えば氷などが多いようです。

荷物を押す（滑らせる）ためには、まず、キャラクターを荷物に隣接させます（Fig. 1-17）。さらに荷物の方向にレバーを入力して、キャラクターが荷物の方を向くようにします。この状態でボタンを押すと、荷物が滑り始めます（Fig. 1-18）。ボタンを押さなければ、レバーを入力しても荷物は動きません。

移動方向に障害物がないかぎり、荷物は自動的に滑り続けます（Fig. 1-19）。レバーやボタンの操作は必要ありません。

壁に衝突すると、荷物は自動的に停止します（Fig. 1-20）。他の荷物に衝突した場合も、荷物は停止します（Fig. 1-21）。

滑る荷物を押すアクションは『ペンゴ』などに採用されています。このゲームでは、ステージに配置された氷を滑らせ、敵をつぶして倒すことができます。アクション中心のゲームですが、特別な氷を3つ並べるとボーナス点が入る、というパズルゲーム風の要素も盛り込まれています。

Fig. 1-17 荷物の方を向く

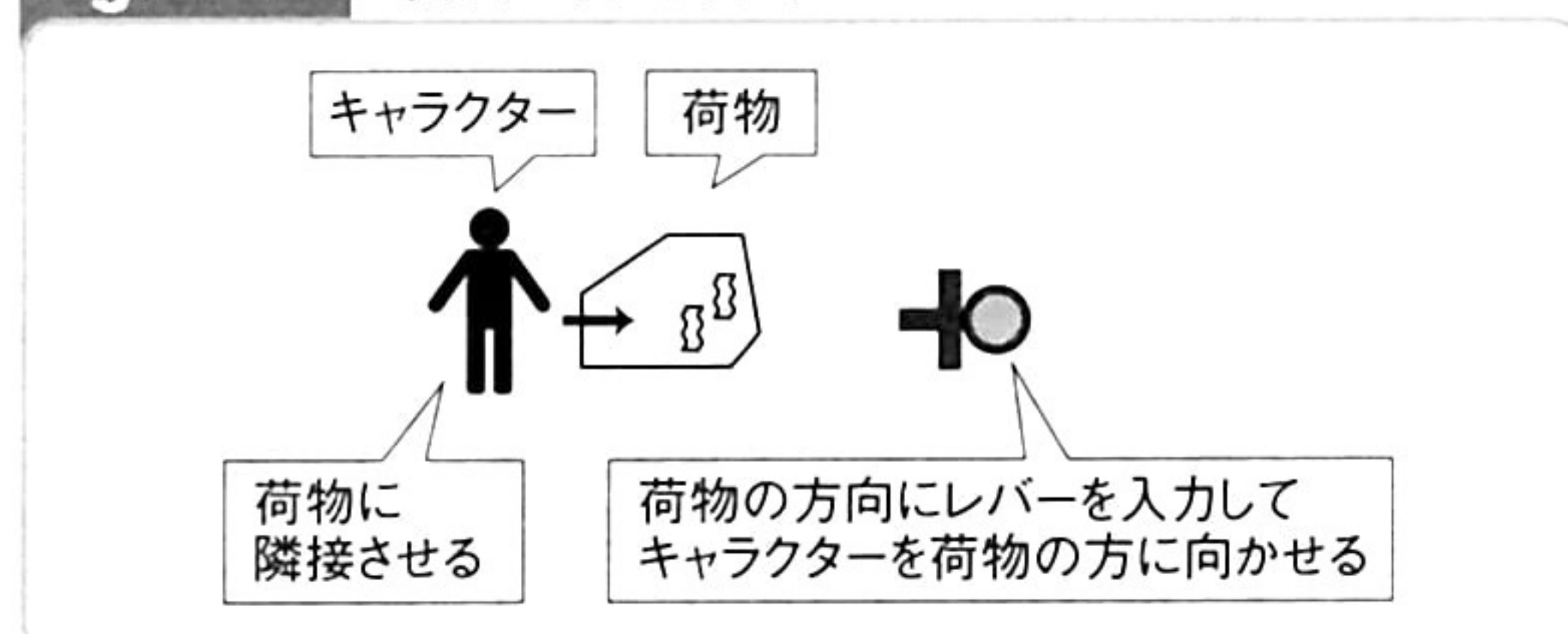



Fig. 1-18 荷物を滑らせる

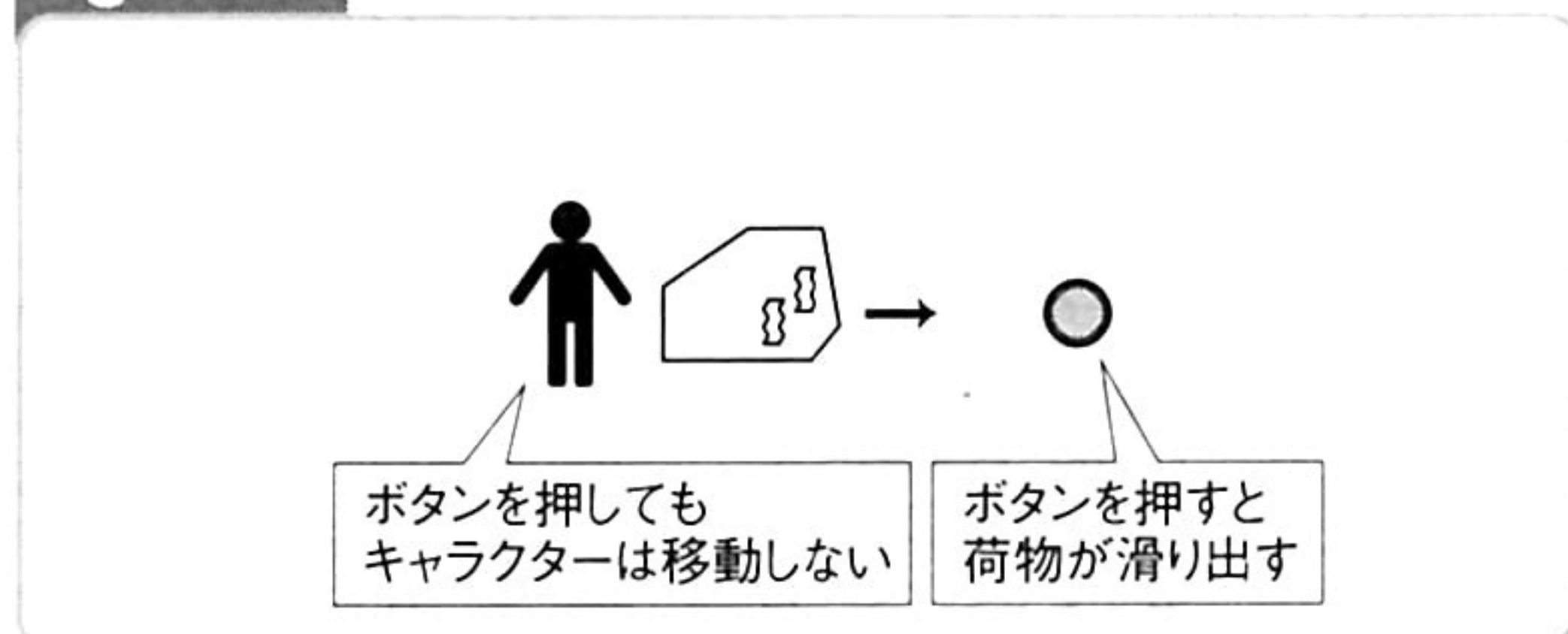



Fig. 1-19 荷物は自動的に滑り続ける

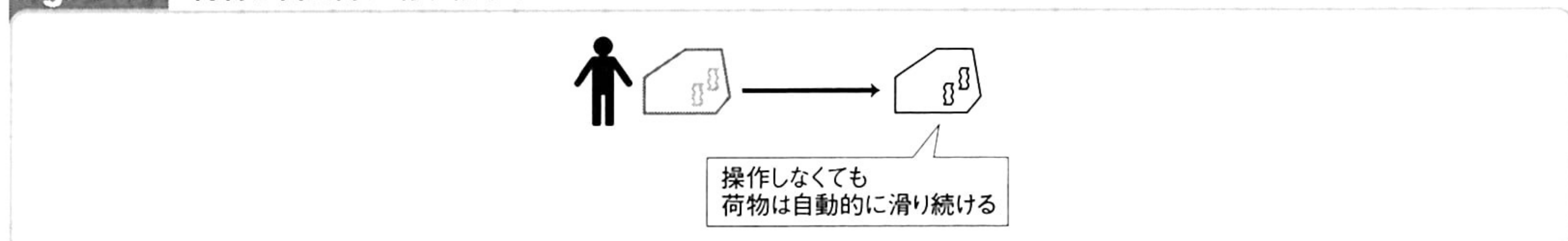



Fig. 1-20 壁に衝突すると停止する

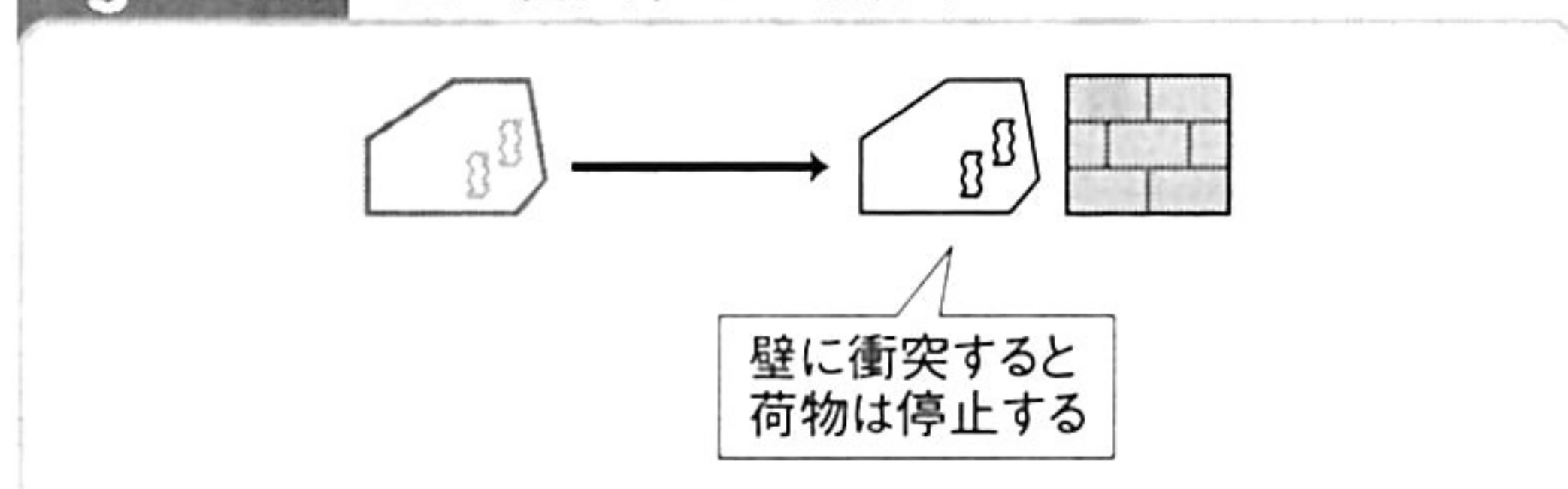
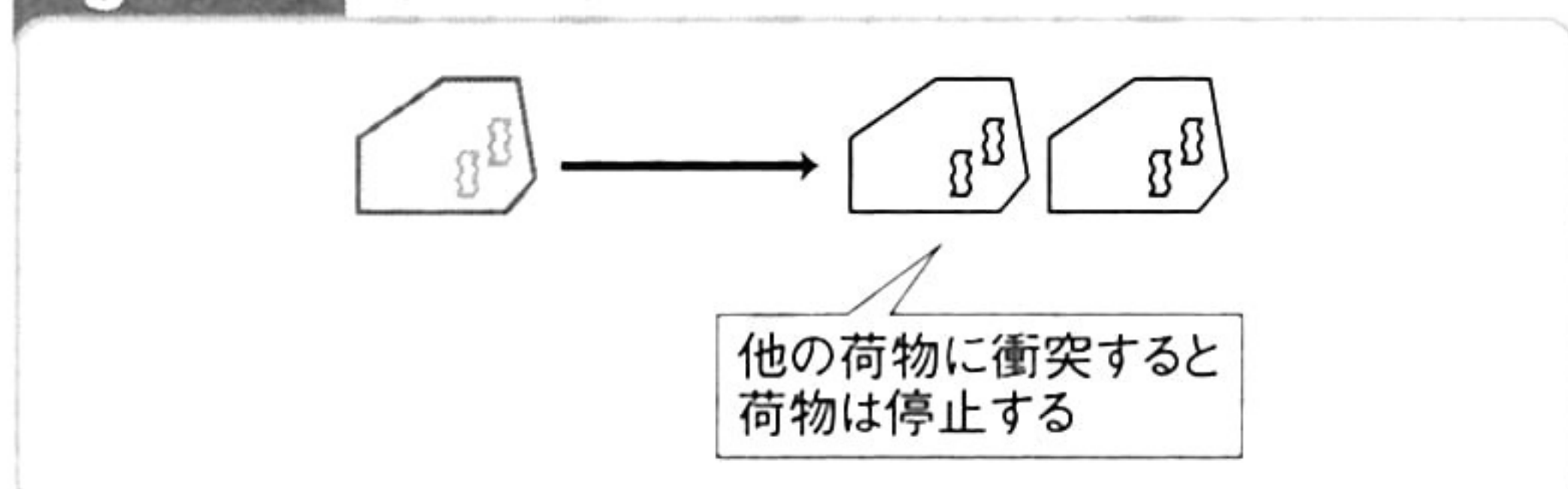



Fig. 1-21 他の荷物に衝突すると停止する

アルゴリズム

キャラクターが荷物を右向きに押そうとしている状況を考えます（Fig. 1-22）。キャラクターが、右にある荷物の方を向いていて、ボタンを入力した場合です。

この状態をセルで表現すると、Fig. 1-23のようになります。キャラクターを文字「M」(Man)、荷物を文字「L」(Load)、壁を文字「#」で示しました。

滑る荷物を実現するには、荷物の移動先のセルが空かどうかを調べます（Fig. 1-24）。移動先が空の場合には荷物を滑らせます。そして、荷物の移動元と移動先のセルを更新します（Fig. 1-25）。

移動先が空であるかぎり、荷物の移動を続けます（Fig. 1-26）。これで荷物が自動的に滑って

Fig. 1-22 荷物を押す

荷物の方を向いて
ボタンを入力する

Fig. 1-23 荷物を押すときのセルの状態

壁のセル
(#)
キャラクターのセル
(M)
荷物のセル
(L)

Fig. 1-24 移動先のセルを調べる

移動先のセルが
空かどうかを調べる

Fig. 1-25 セルの更新

移動元のセルを
空にする
移動先のセルを
荷物にする

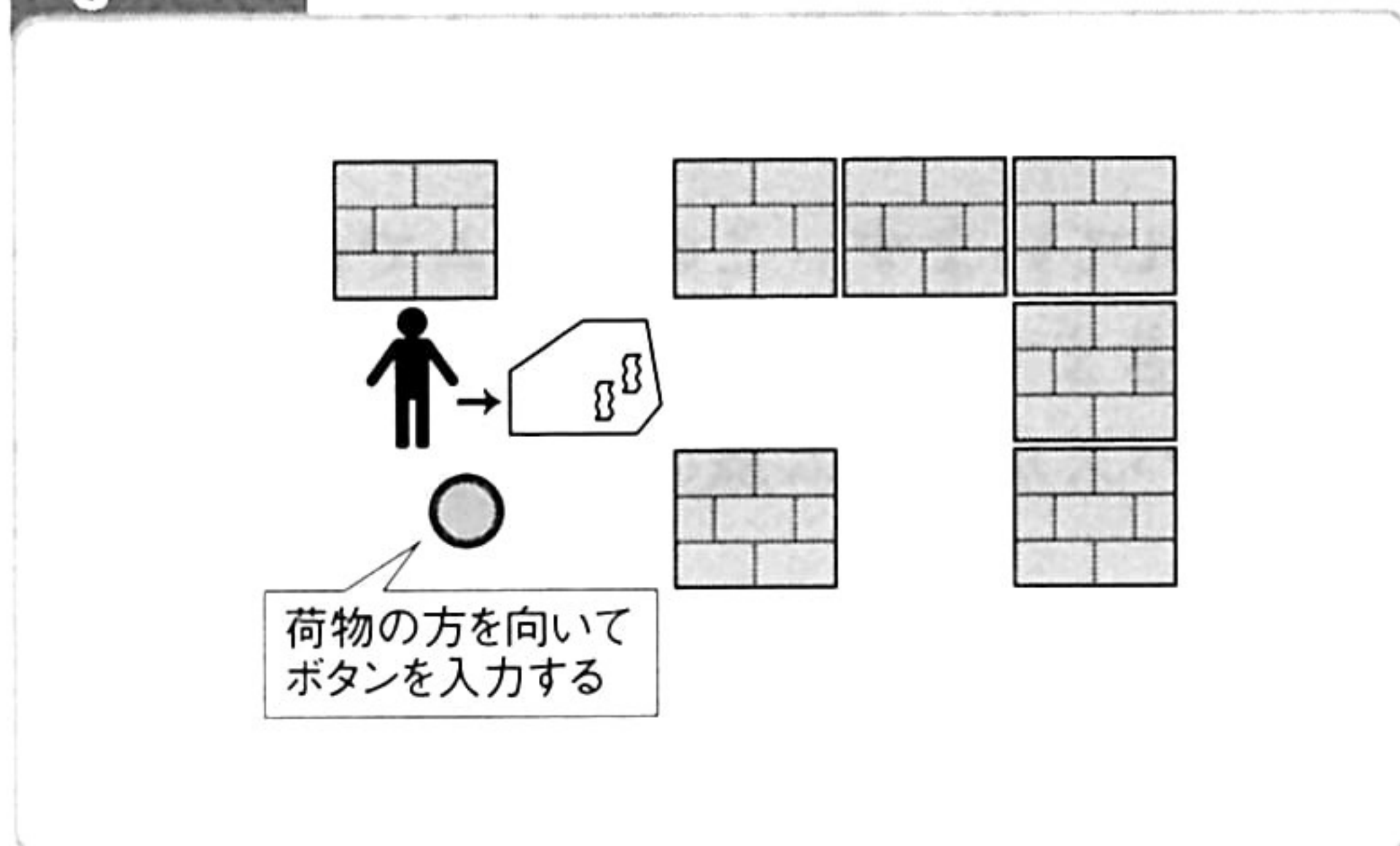
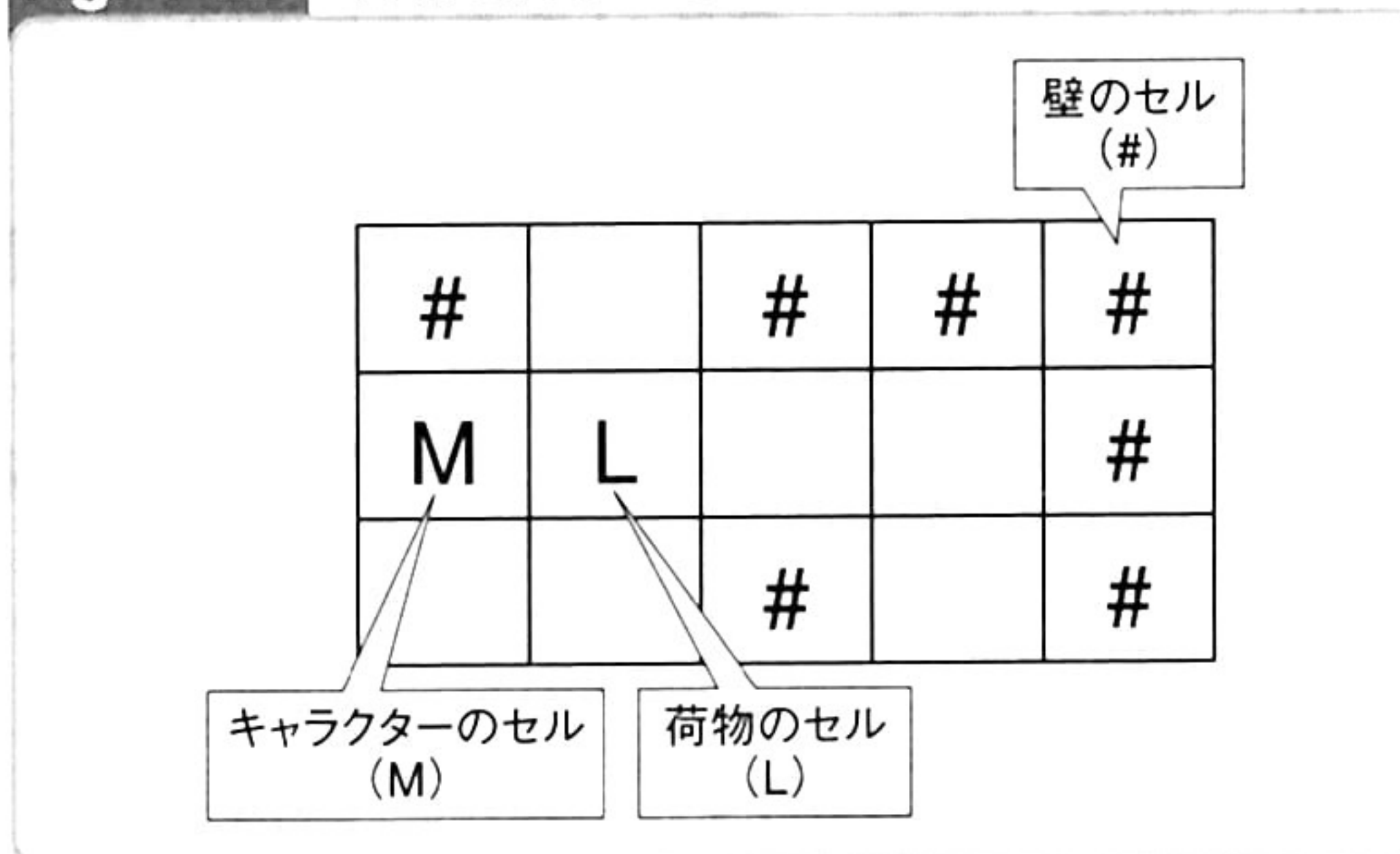
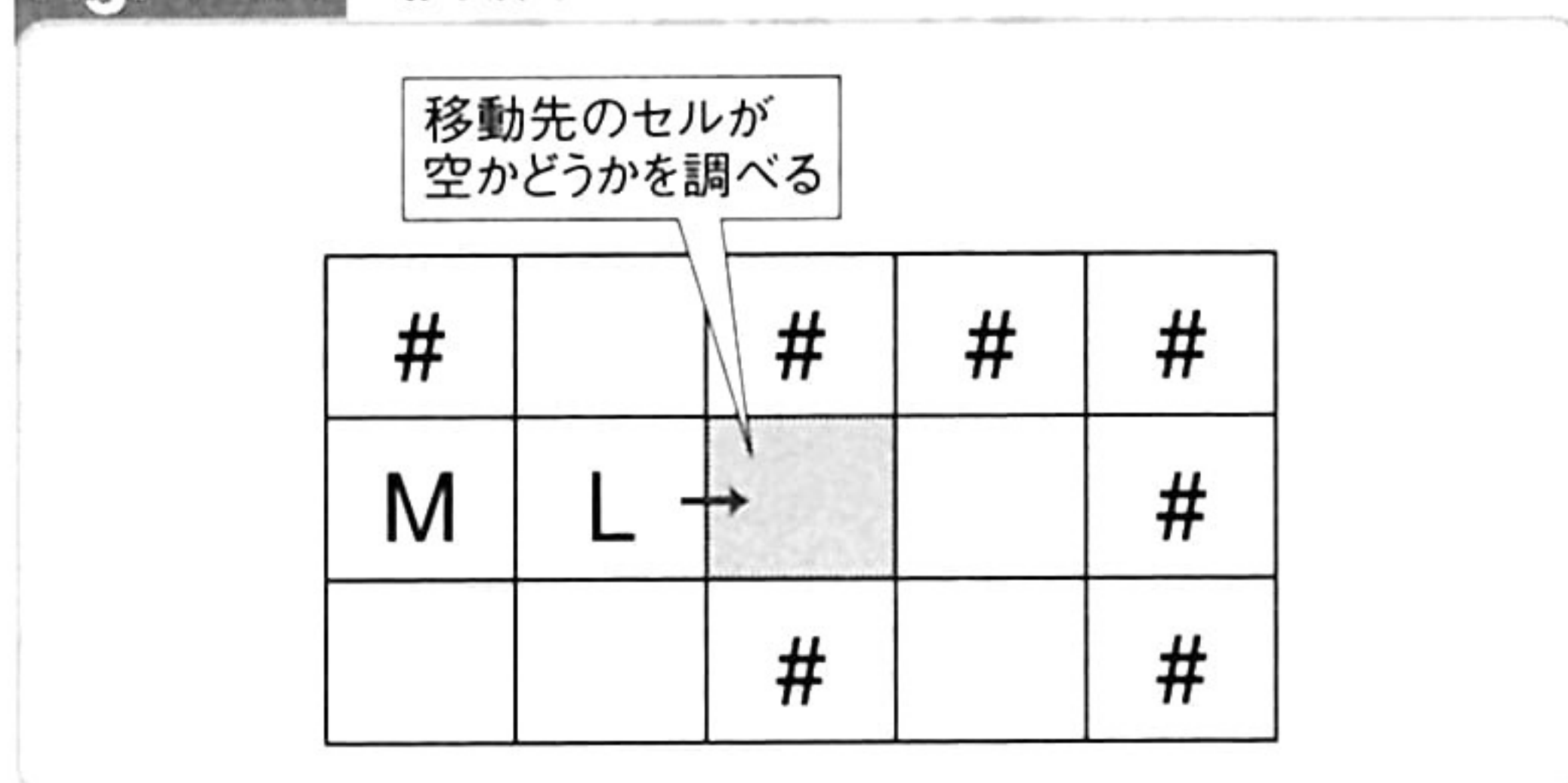
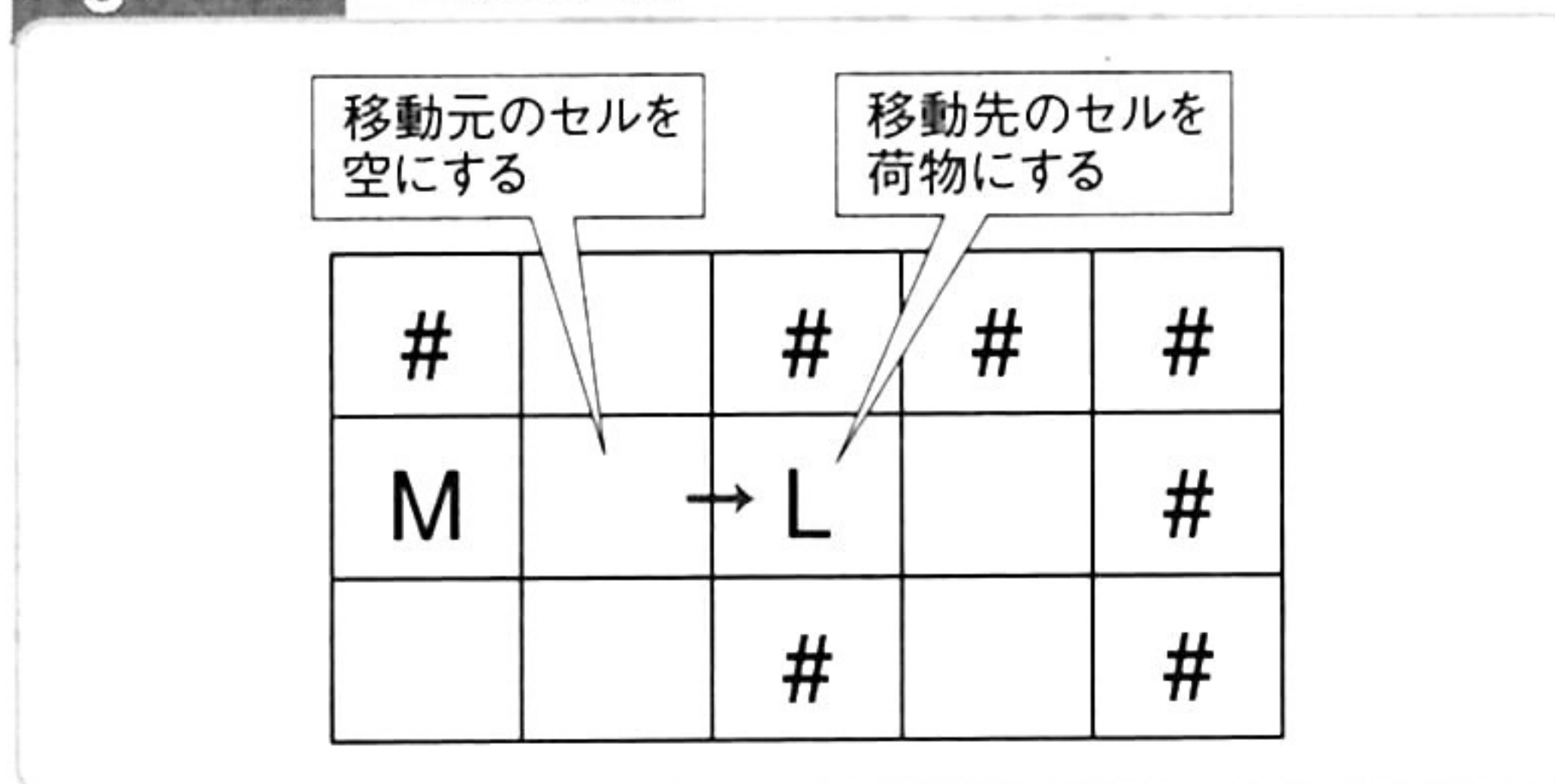
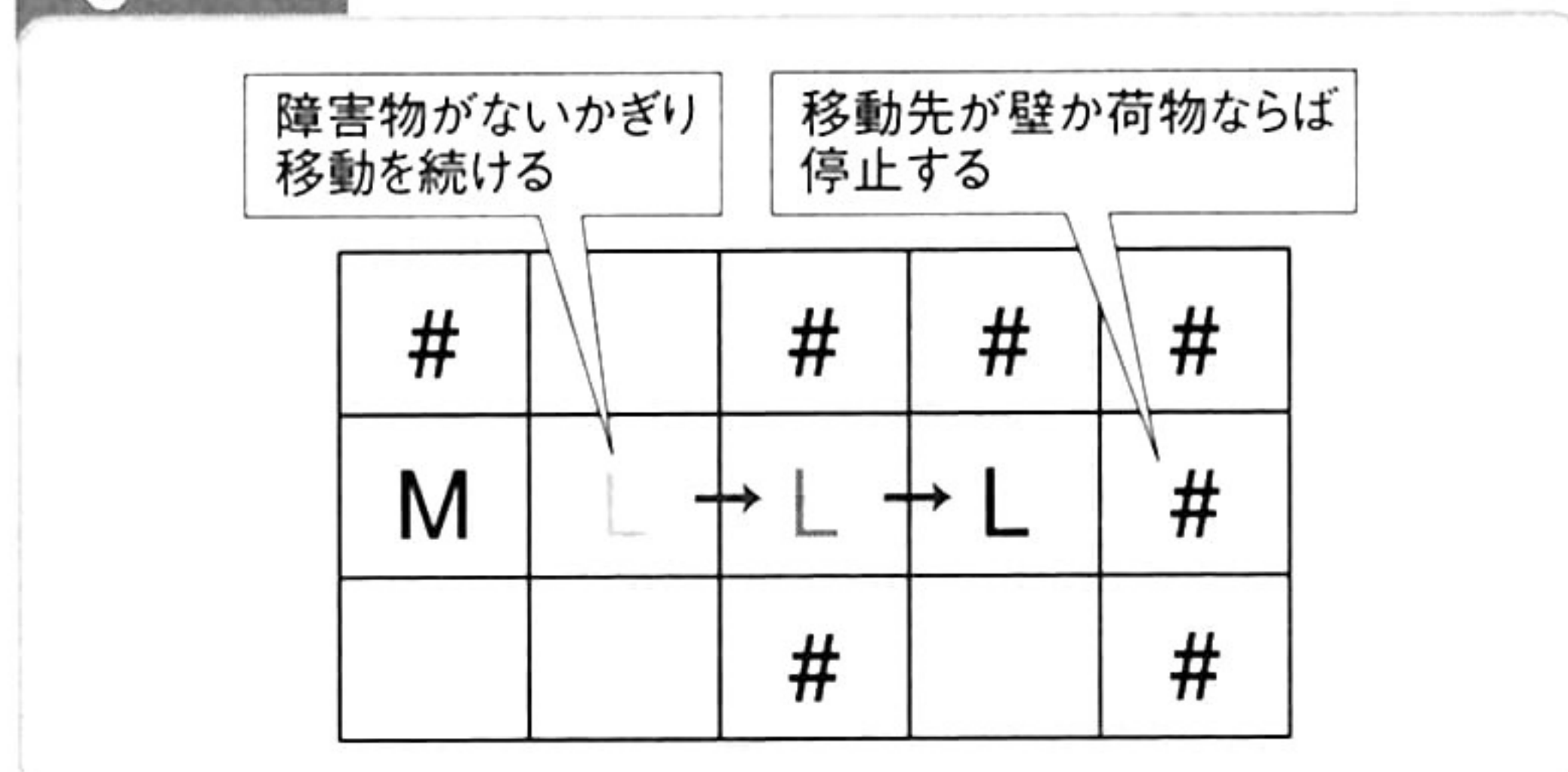
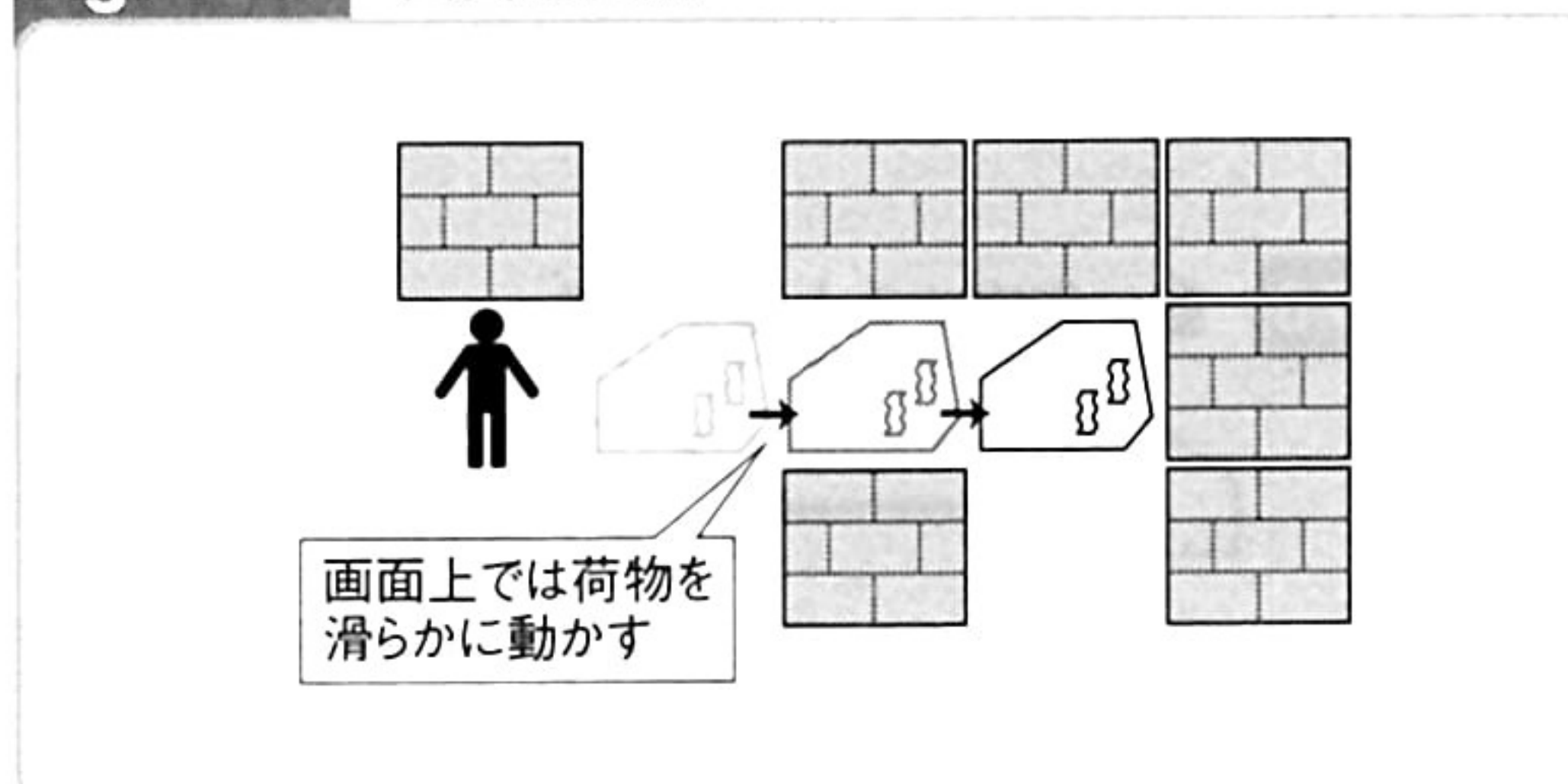



Fig. 1-26 移動先が空であるかぎり移動を続ける

障害物がないかぎり
移動を続ける
移動先が壁か荷物ならば
停止する

Fig. 1-27 画面の更新

画面上では荷物を
滑らかに動かす

いく動きになります。荷物の移動先が壁の場合、あるいは他の荷物の場合には、荷物を停止します。

セルの更新に合わせて、画面も更新します (Fig. 1-27)。荷物の描画座標を少しずつ変化させることによって、荷物が滑らかに動いているように見せます。

プログラム

List 1-3は滑る荷物を押すプログラムです。荷物とキャラクターの移動処理、および荷物を押すときの処理を掲載しました。

プログラムの構成は「荷物を押す」(→p. 10) とほぼ同じですが、荷物の移動処理は大きく違っています。滑る荷物の場合には、荷物の移動処理において、荷物がまだ移動できるかどうかの判定を行います。荷物の前方に障害物がないかぎり、移動を継続します。

List 1-3　滑る荷物を押す (CSlidingLoadPusherLoadクラス、CSlidingLoadPusherManクラス)

```
// 荷物を押すときに呼び出す処理
void CSlidingLoadPusherLoad::Push(int vx, int vy) {

    // 静止状態から移動状態に移行する
    if (State==0) {

        // 移動方向の設定
        VX=vx;
        VY=vy;

        // タイマーと状態の設定
        Time=1;
        State=1;
    }
}

// 荷物の移動処理
bool CSlidingLoadPusherLoad::Move(const CInputState* is) {

    // 移動状態の処理
    if (State==1) {

        // タイマーの更新
        Time--;

        // 描画座標の更新
        X=CX-VX*Time*0.2f;
        Y=CY-VY*Time*0.2f;

        // タイマーが0になったときの処理
```

```
        if (Time==0) {

            //移動先のセルが空かどうかを調べる
            if (Cell->Get(CX+VX, CY+VY)==' ') {

                //移動先が空のときには移動を続ける
                //移動元と移動先のセルを更新する
                Cell->Set(CX, CY, ' ');
                Cell->Set(CX+VX, CY+VY, 'L');

                //セル座標の更新
                CX+=VX;
                CY+=VY;

                //タイマーの設定
                Time=5;
            } else {

                //移動先が空ではないときには停止する
                //静止状態に移行する
                State=0;
            }
        }
    }
    return true;
}

//キャラクターの移動処理
bool CSlidingLoadPusherMan::Move(const CInputState* is) {

    //静止状態の処理
    if (State==0) {

        //レバー入力に応じて移動方向を設定する
        if (is->Left) {
            VX=-1; VY=0;
        } else
        if (is->Right) {
            VX=1; VY=0;
        } else
        if (is->Up) {
            VY=-1; VX=0;
        } else
        if (is->Down) {
            VY=1; VX=0;
        }

        //レバー入力があった場合の処理
        if (is->Left || is->Right || is->Up || is->Down) {
```

```
			// 移動先のセルが空かどうかを調べて、
			// 空の場合には移動する
			if (Cell->Get(CX+VX, CY+VY)==' ') {

				// 移動元と移動先のセルを更新する
				Cell->Set(CX, CY, ' ');
				Cell->Set(CX+VX, CY+VY, 'M');

				// セル座標の更新
				CX+=VX;
				CY+=VY;

				// 状態とタイマーの設定
				// 移動状態に移行する
				State=1;
				Time=10;
			}
		}

		// 荷物に向かってボタンを押したときの処理
		if (
			is->Button[0] &&
			Cell->Get(CX+VX, CY+VY)=='L'
		) {

			// 荷物のオブジェクトを探す
			for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
				CMover* mover=(CMover*)i.Next();

				// 荷物を押す処理を呼び出す
				if ((int)mover->X==CX+VX && (int)mover->Y==CY+VY) {
					((CSlidingLoadPusherLoad*)mover)->Push(VX, VY);
				}
			}
		}
	}

	// 移動状態の処理
	if (State==1) {

		// タイマーの更新
		Time--;

		// 描画座標の更新
		X=CX-VX*Time*0.1f;
		Y=CY-VY*Time*0.1f;

		// タイマーが0になったら静止状態に移行する
```

```
            if (Time==0) {
                State=0;
            }
        }
        return true;
    }
```

SAMPLE

「SLIDING LOAD PUSHER」は「滑る荷物を押す」のサンプルです。レバーの上下左右(カーソルキーの上下左右)でキャラクターが移動します。

荷物の方を向いてからボタン0(Zキー)を押すと、荷物を滑らせて動かすことができます。荷物は自動的に滑っていき、他の荷物や壁にぶつかると止まります。

SLIDING LOAD PUSHER → p. 386

重力で落ちる荷物を押す

これも、キャラクターが荷物を押すアクションです。「荷物を押す」(→p. 10) に似ていますが、こちらは画面を横から見た形になり、下に壁や他の荷物がないときには、荷物は重力に引かれて落ちていきます。

荷物を押すには、キャラクターを荷物に隣接させて、荷物を押すようにキャラクターを動かします (Fig. 1-28)。荷物の移動先に壁や他の荷物がなければ、荷物を押すことができます。

しかし、荷物の下に壁や他の荷物がないときには、荷物は重力に引かれて落下します (Fig. 1-29)。壁や他の荷物の上に載るまで、落下は続きます。

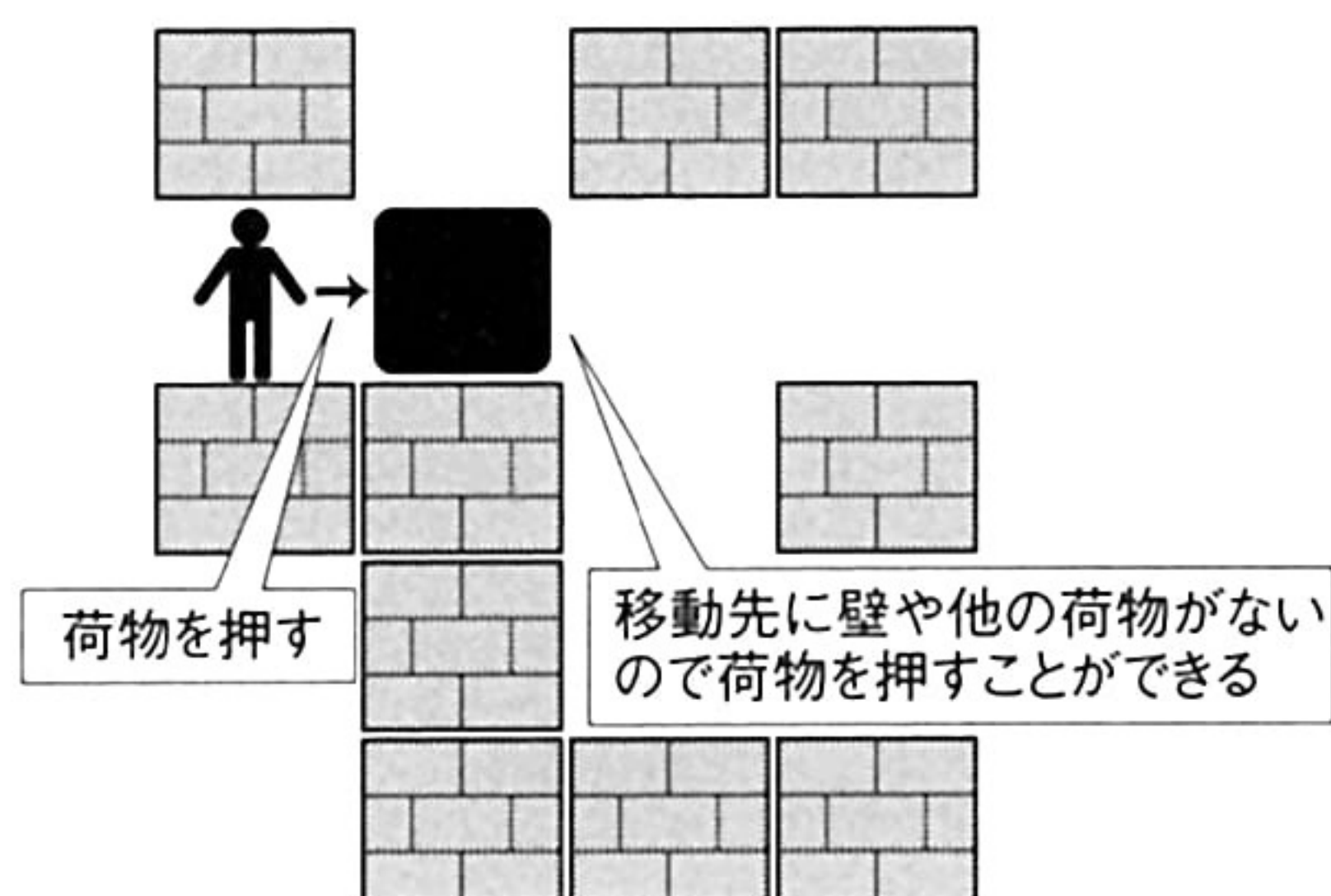
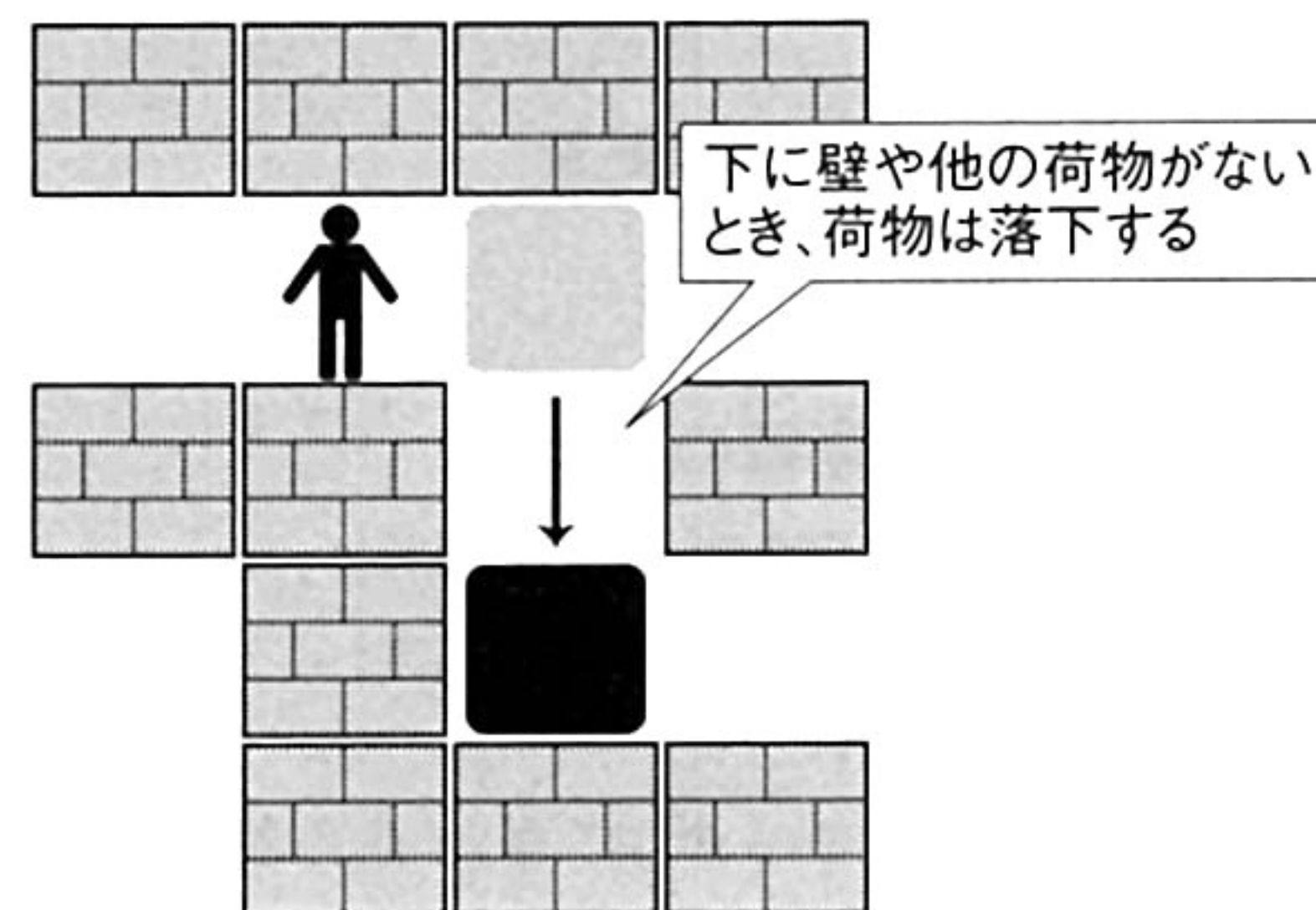
落下する荷物を押すアクションは、数多くのアクションゲームやパズルゲームに採用されています。例えば『フラッピー』は、落下する荷物を目的地まで運ぶことが目的のパズルゲーム

Fig. 1-28 荷物を押す

荷物を押す

移動先に壁や他の荷物がないので荷物を押すことができる

Fig. 1-29 落下する荷物

下に壁や他の荷物がないとき、荷物は落下する

です。他の荷物を積み重ねて輸送のルートを作ったり、ジャマな敵に荷物を落として押しつぶしたりと、いろいろな要素が盛り込まれています。

アルゴリズム

落下する荷物は「荷物を押す」(→p. 10) の応用で実現できます。キャラクターが荷物を押した状況をセルで表現してみましょう (Fig. 1-30)。

荷物を移動するには、移動先のセルが空かどうかを調べます。セルが空の場合には、移動前と移動後のセルを更新して、荷物を移動します (Fig. 1-31)。

次に、荷物の下方向にあるセルを調べます (Fig. 1-32)。セルが空の場合には、荷物を落下させます (Fig. 1-33)。荷物の下に壁や他の荷物がないかぎり、落下を続けます。

荷物を細かく動かす

ちなみに『フラッピー』では、荷物をより細かい単位で移動することができます。荷物の半分の幅だけずらすことができるため、工夫しだいでは少ない荷物の上に多数の荷物を積むこと

Fig. 1-30 荷物を押すときのセルの状態

#	#	#	#
M →	L		
#	#		#
	#		
	#	#	#

Fig. 1-31 セルの更新

#	#	#	#
	M	L	
#	#		#
	#		
	#	#	#

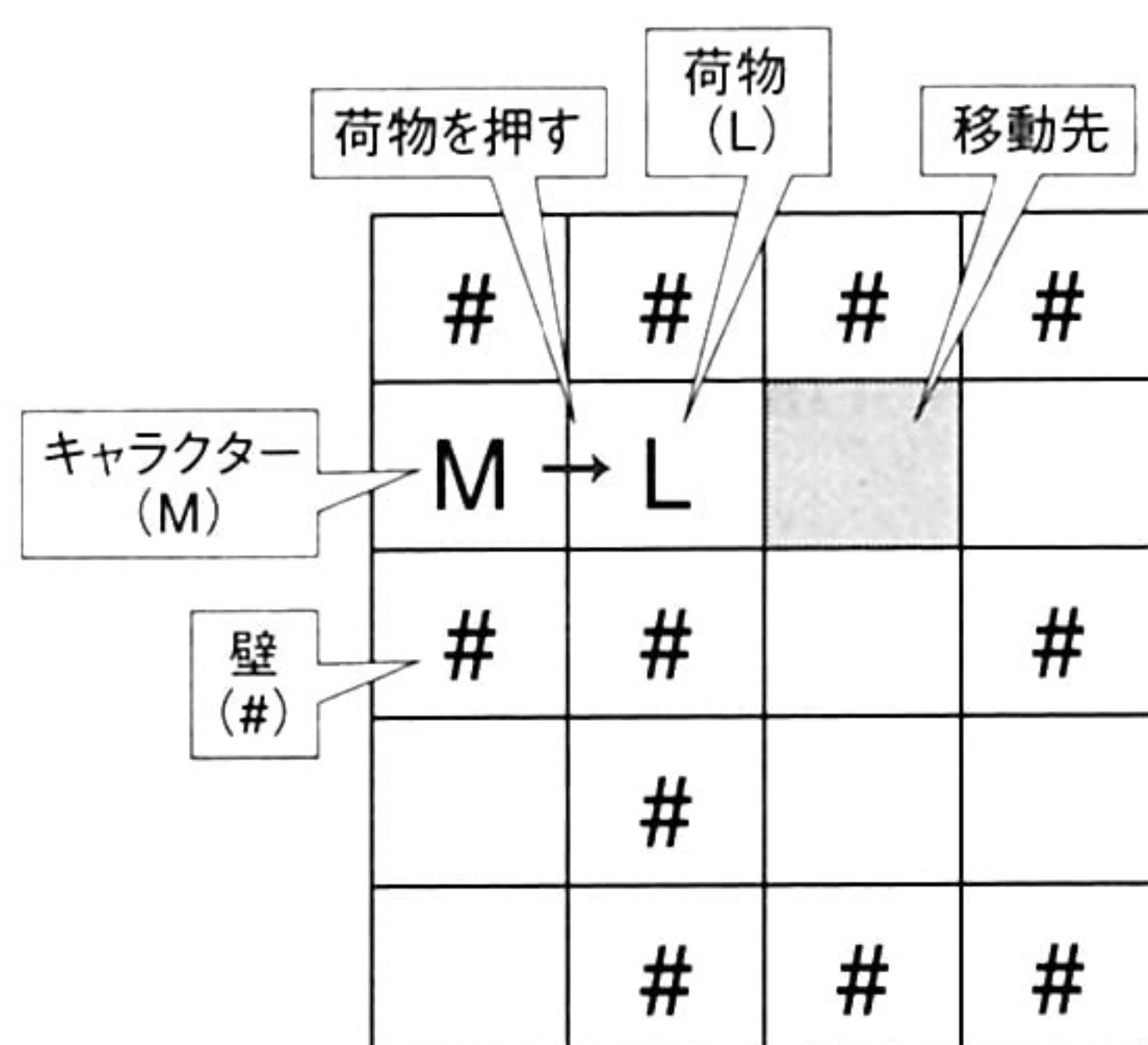
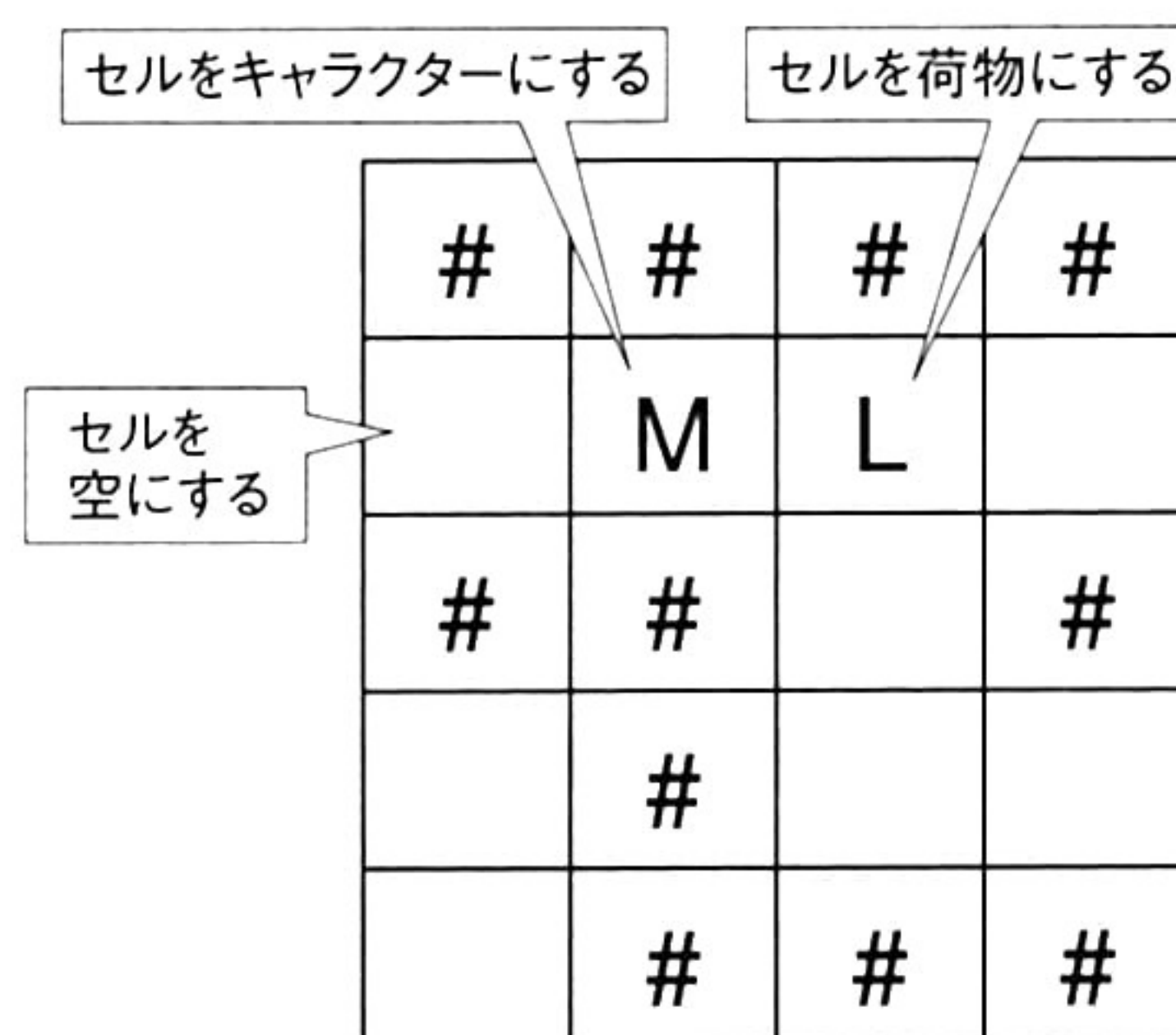



Fig. 1-32 荷物の下方向にあるセルを調べる

#	#	#	#
	M	L	
#	#		#
	#		
	#	#	#

Fig. 1-33 荷物を落下させる

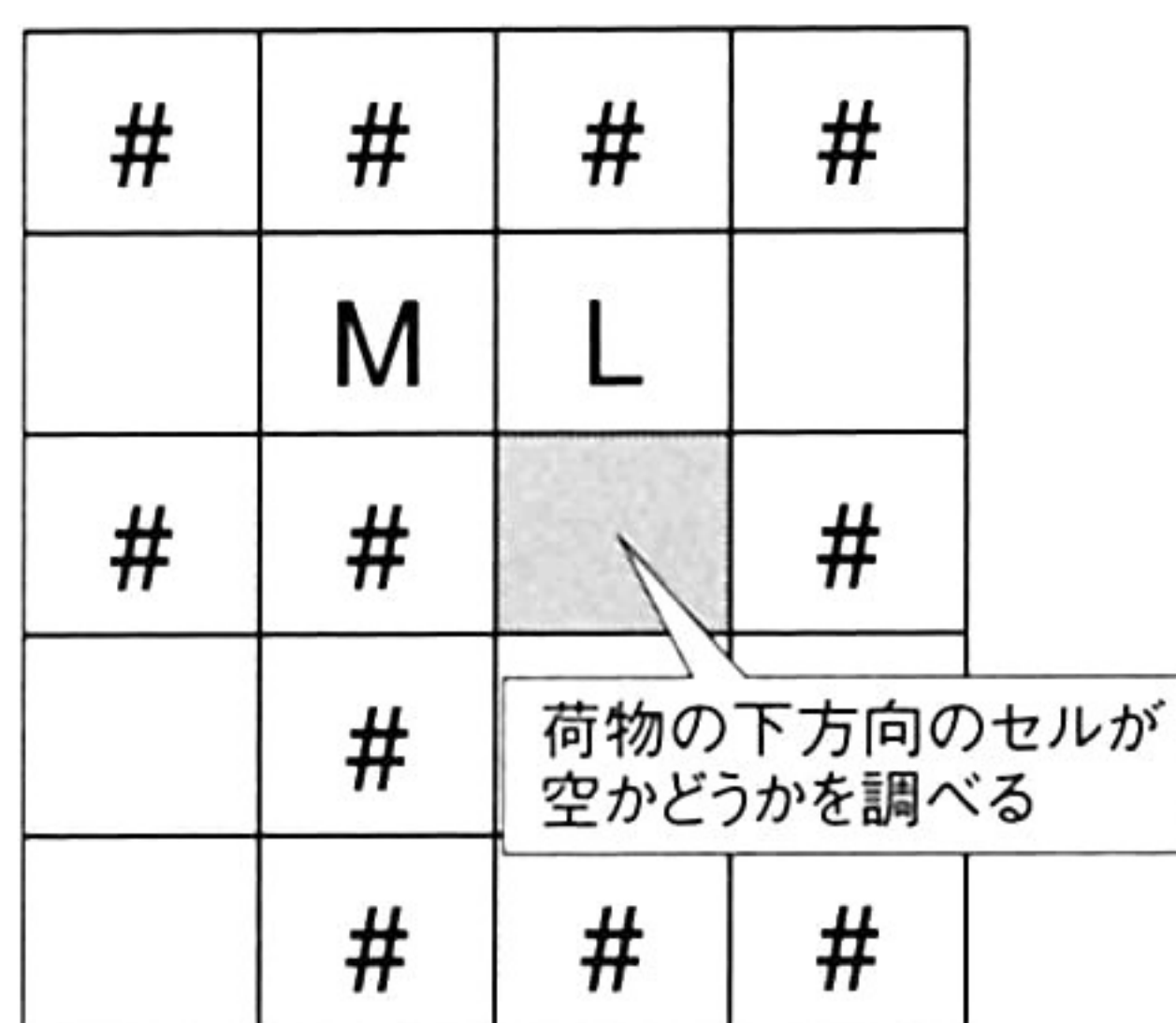
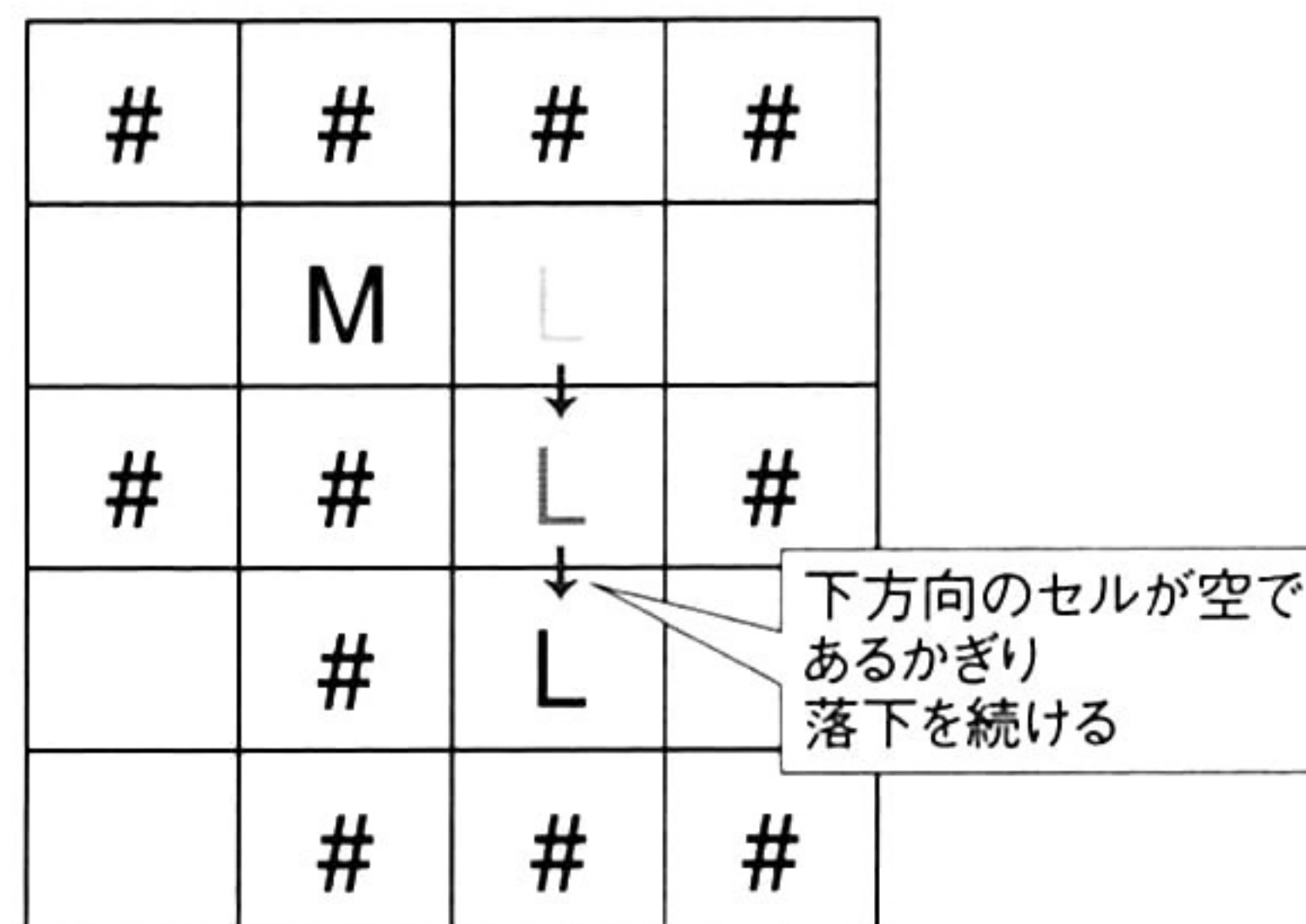
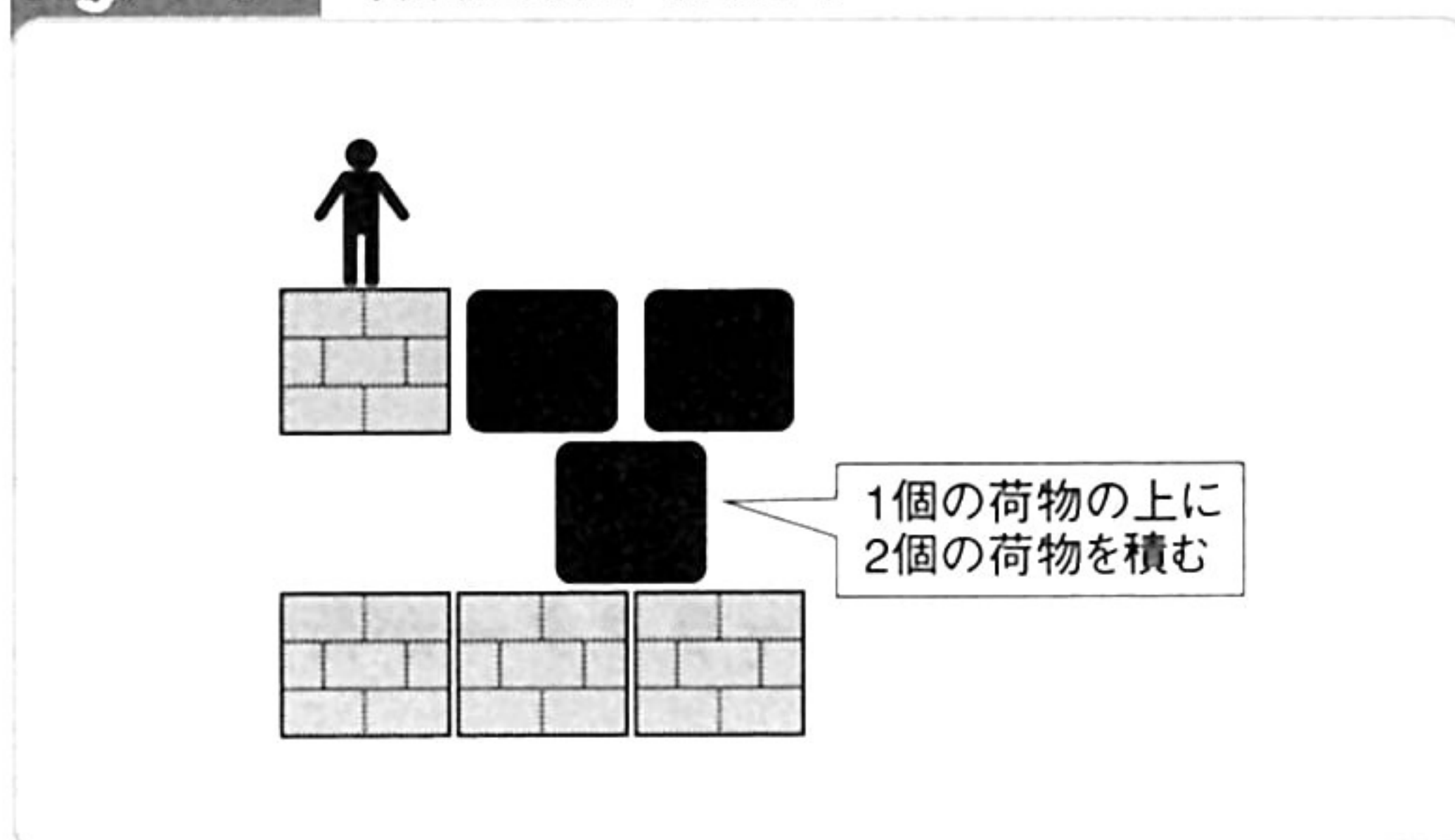
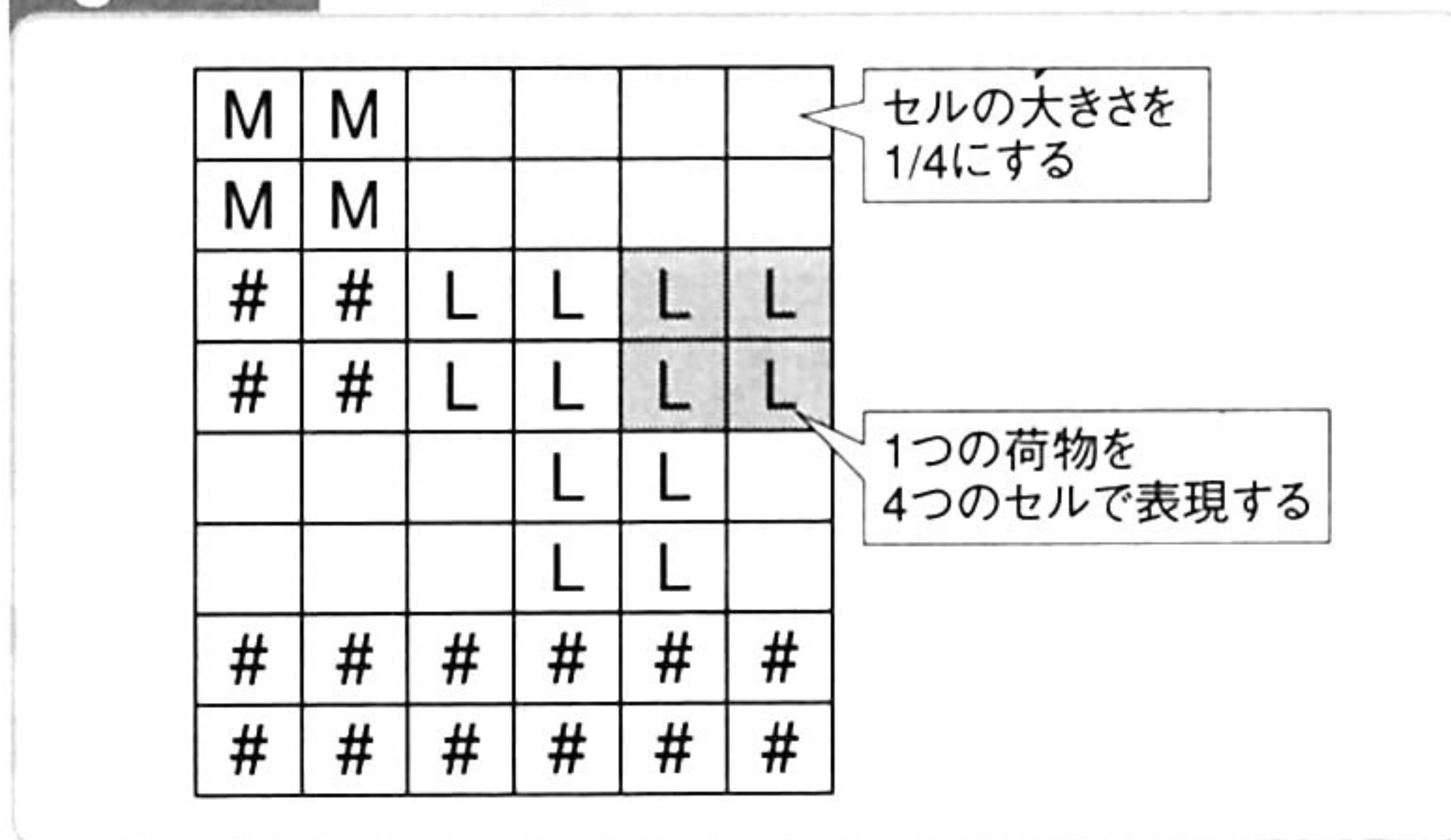
#	#	#	#
	M	L ↓	
#	#	L ↓	#
	#	L	
	#	#	#

Fig. 1-34 荷物を細かく動かす

1個の荷物の上に
2個の荷物を積む

Fig. 1-35 セルを細かくする

M	M				
M	M				
#	#	L	L	L	L
#	#	L	L	L	L
			L	L	
			L	L	
#	#	#	#	#	#
#	#	#	#	#	#

セルの大きさを
1/4にする

1つの荷物を
4つのセルで表現する

も可能です(Fig. 1-34)。この要素がゲームの面白さを深めています。

荷物をより細かい単位で移動したい場合には、セルを細かくします。例えばセルを1/4の大きさにして、1つの荷物を4つのセルで表現する方式にすれば、荷物の半分の幅を単位にして縦横に動かすことができます(Fig. 1-35)。

プログラム

List 1-4は重力で落ちる荷物を押すプログラムです。荷物とキャラクターの移動処理、および荷物を押すときの処理を掲載しました。

荷物を押す処理は「荷物を押す」(→p. 10)によく似ています。違うのは荷物の移動処理です。重力で落ちる荷物の場合には、荷物の下方向にあるセルを調べて、空のときには落下します。壁や他の荷物に衝突しないかぎり、落下を続けます。これは、「滑る荷物を押す」(→p. 25)の処理に似ています。

List 1-4 重力で落ちる荷物を押す(CLoadPusherInGravityLoadクラス、CLoadPusherInGravityManクラス)

```
// 荷物を押すときに呼び出す処理
void CLoadPusherInGravityLoad::Push(int vx, int vy) {

    // 静止状態ならば移動状態に移行する
    if (State==0) {

        // 移動方向の設定(左右方向)
        VX=vx;
        VY=0;

        // セル座標の更新
        CX+=vx;
        CY+=vy;

        // 状態とタイマーの設定
```

```
            State=1;
            Time=10;
        }
    }

//荷物の移動処理
bool CLoadPusherInGravityLoad::Move(const CInputState* is) {

    //静止状態の処理
    if (State==0) {

        //下方向のセルが空ならば、移動状態に移行する
        if (Cell->Get(CX, CY+1)==' ') {

            //移動方向の設定(下方向)
            VX=0;
            VY=1;

            //セルの更新
            Cell->Set(CX, CY, ' ');
            Cell->Set(CX, CY+VY, 'L');

            //セル座標の更新
            CX+=VX;
            CY+=VY;

            //状態とタイマーの設定
            //移動状態に移行する
            State=1;
            Time=10;
        }
    }

    //移動状態の処理
    if (State==1) {

        //タイマーの更新
        Time--;

        //描画座標の更新
        X=CX-VX*Time*0.1f;
        Y=CY-VY*Time*0.1f;

        //タイマーが0になったら静止状態に移行する
        if (Time==0) {
            State=0;
        }
    }
    return true;
```

```
}

// キャラクターの移動処理
bool CLoadPusherInGravityMan::Move(const CInputState* is) {

    // 静止状態の処理
    if (State==0) {

        // レバー入力に応じて移動方向を設定する
        VX=VY=0;
        if (is->Left) VX=-1; else
        if (is->Right) VX=1; else
        if (is->Up) VY=-1; else
        if (is->Down) VY=1;

        // 移動する場合の処理
        if (VX!=0 || VY!=0) {

            // 移動先のセルが空ならば移動する
            if (Cell->Get(CX+VX, CY+VY)==' ') {

                // セルの更新
                Cell->Set(CX, CY, ' ');
                Cell->Set(CX+VX, CY+VY, 'M');

                // セル座標の更新
                CX+=VX;
                CY+=VY;

                // 状態とタイマーの設定
                // 移動状態に移行する
                State=1;
                Time=10;
            } else

            // 移動先のセルが荷物で、
            // かつ荷物の移動先が空ならば、
            // 荷物を押す
            if (
                Cell->Get(CX+VX, CY)=='L' &&
                Cell->Get(CX+VX*2, CY)==' ' &&
                Cell->Get(CX+VX, CY+1)!=' '
            ) {
                // セルの更新
                Cell->Set(CX, CY, ' ');
                Cell->Set(CX+VX, CY, 'M');
                Cell->Set(CX+VX*2, CY, 'L');

                // セル座標の更新
```

```
                    CX+=VX;
                    CY+=VY;

                    // 状態とタイマーの設定
                    State=1;
                    Time=10;

                    // 荷物のオブジェクトを探す
                    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
                        CMover* mover=(CMover*)i.Next();

                        // 荷物を押す処理を呼び出す
                        if ((int)mover->X==CX && (int)mover->Y==CY) {
                            ((CLoadPusherInGravityLoad*)mover)->Push(VX, VY);
                        }
                    }
                }
            }
        }

        // 移動状態の処理
        if (State==1) {

            // タイマーの更新
            Time--;

            // 描画座標の更新
            X=CX-VX*Time*0.1f;
            Y=CY-VY*Time*0.1f;

            // タイマーが0になったら静止状態に移行する
            if (Time==0) {
                State=0;
            }
        }
        return true;
    }
```

SAMPLE

「LOAD PUSHER IN GRAVITY」は「重力で落ちる荷物を押す」のサンプルです。レバーの上下左右（カーソルキーの上下左右）でキャラクターが移動します。

荷物に向かってキャラクターを動かすと、荷物を左右に押して動かすことができます。荷物の下に他の荷物や壁がないと、荷物は落下します。

LOAD PUSHER IN GRAVITY → p. 386

自律的に動くキャラクター

レバーやボタンなどを入力しなくても、ステージ内を自動的に動き回るキャラクターです。プレイヤーはキャラクターを直接動かすのではなく、「穴を掘れ」や「壁を登れ」といった命令を出します。キャラクターは命令にしたがって、自律的に行動します。

ここでは、歩くだけの「歩行キャラクター」と、歩くことに加えて穴を掘ることもできる「掘削キャラクター」という、2種類のキャラクターを考えましょう。ステージには複数のキャラクターが配置されています（Fig. 1-36）。最初はすべてのキャラクターが歩行キャラクターです。

歩行キャラクター

キャラクターは自律的に行動します。歩行キャラクターの場合には、勝手に左へ右へと歩き回ります（Fig. 1-37）。壁に突き当たったら、移動方向を反転します。自分の下方向に壁がないときには、落下します。

Fig. 1-36 配置されたキャラクター

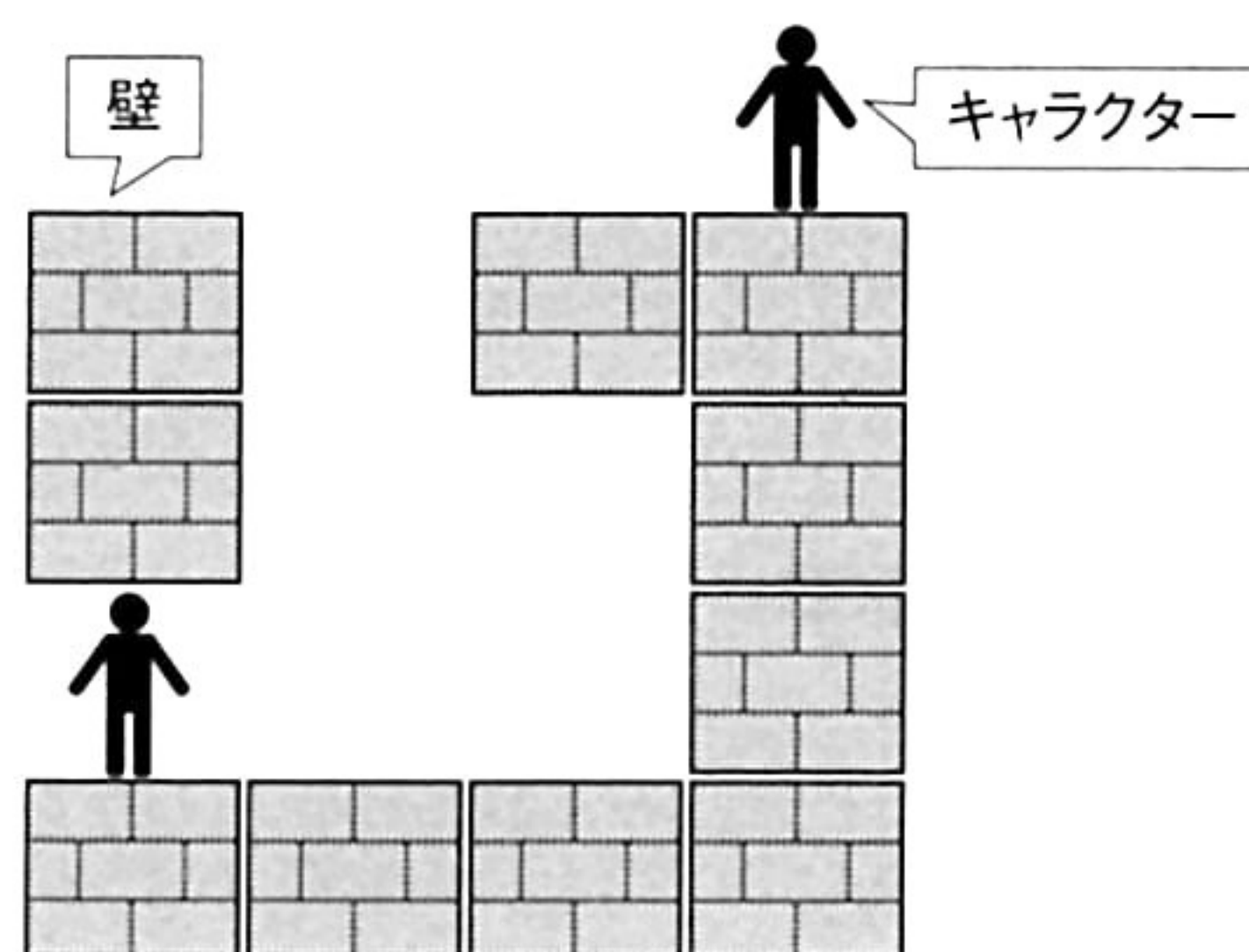



Fig. 1-37 歩行キャラクターの行動

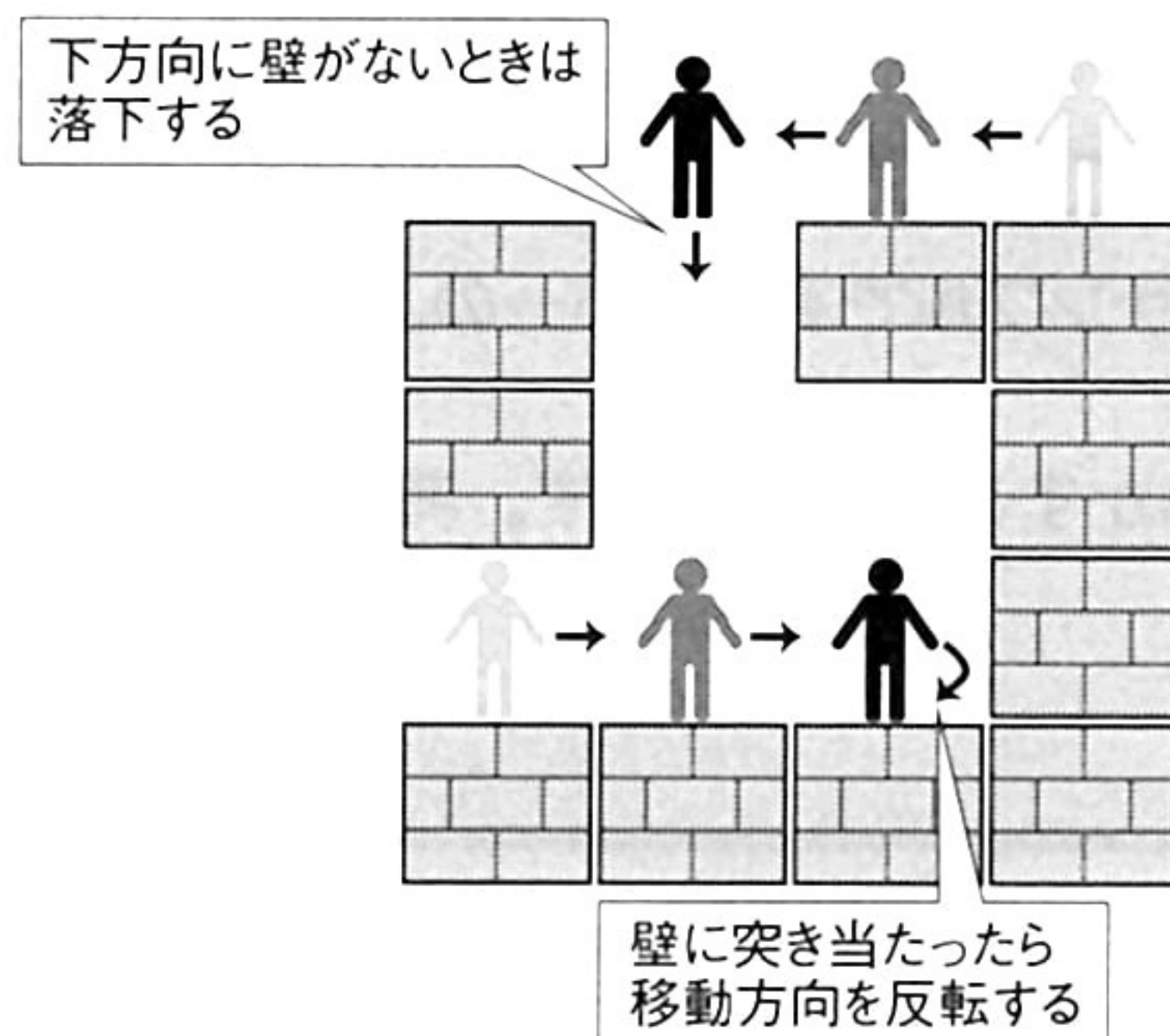
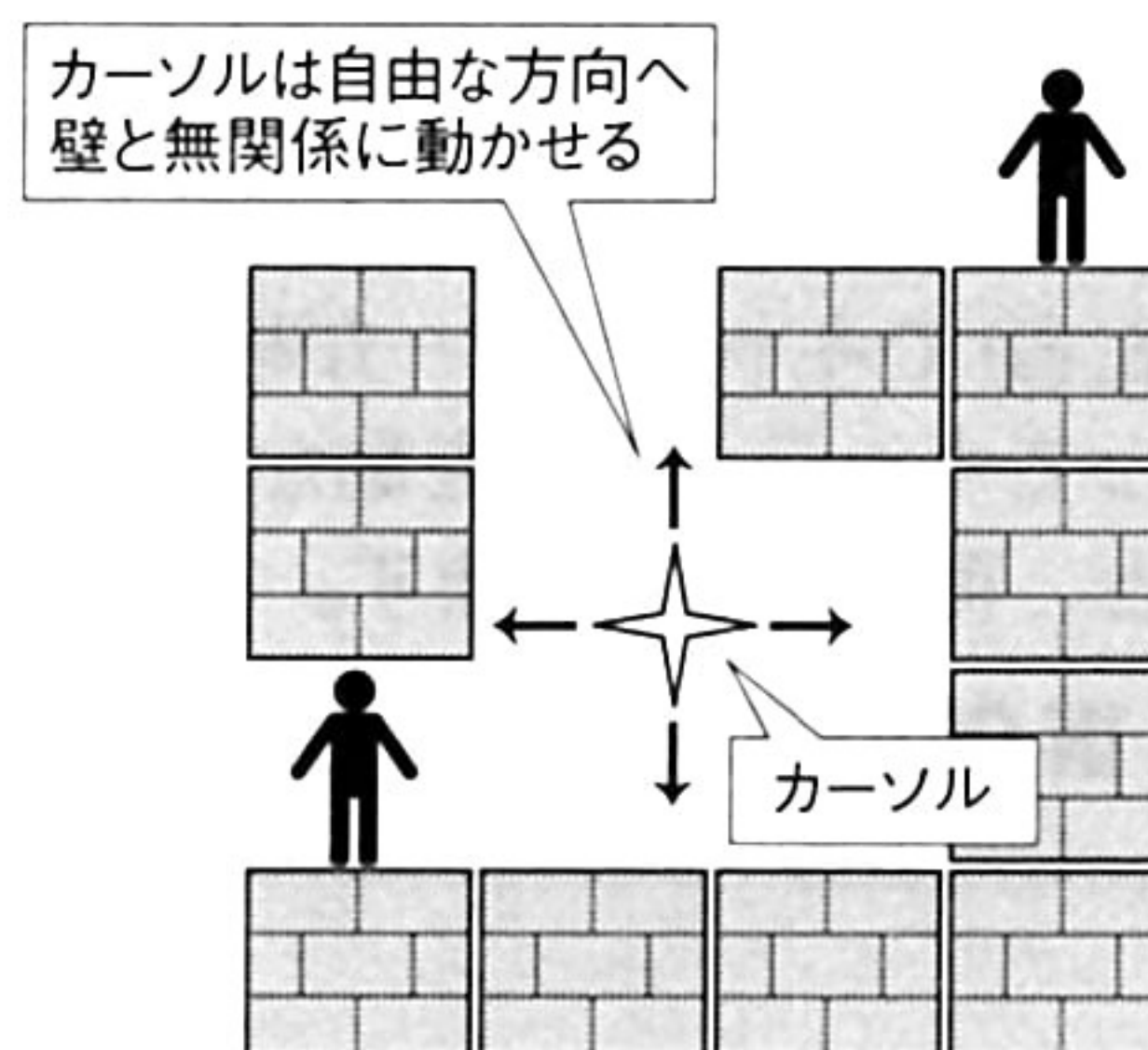



Fig. 1-38 カーソルの操作

プレイヤーはキャラクターを直接操作することができません。かわりに、カーソルを操作することができます (Fig. 1-38)。カーソルは自由な方向へ、壁などの障害物と無関係に動かすことができます。

掘削キャラクター

カーソルを任意のキャラクターに重ねてボタンを押すと、そのキャラクターに命令を与えることができます (Fig. 1-39)。本格的なゲームでは、いくつもの命令のなかから好きなものを選んで発行することができますが、ここでは簡単のために「穴を掘れ」という命令だけを出せることにしましょう。

命令を受けたキャラクターは、掘削キャラクターに変化します (Fig. 1-40)。掘削キャラクターは、自分の下方向にある壁を掘って、穴を空けることができます。

掘削キャラクターが空けた穴は、掘削キャラクターだけではなく、歩行キャラクターも通り抜けることができます (Fig. 1-41)。掘削キャラクターを上手に使えば、壁に囲まれて抜け出せないでいる歩行キャラクターを脱出させることができます。

自律的に動くキャラクターを使ったゲームには『レミングス』などがあります。このゲームでは、ステージ内に非常に多くのキャラクターが登場し、それぞれが命令にしたがって自律的に動き回ります。プレイヤーは適切な命令を与えることによって、キャラクターをゴールまで

Fig. 1-39 キャラクターへの命令

カーソルをキャラクターに重ねて
ボタンを入力すると
命令を与えることができる

Fig. 1-40 掘削キャラクターの行動

掘削キャラクターは
下方向に穴を掘る

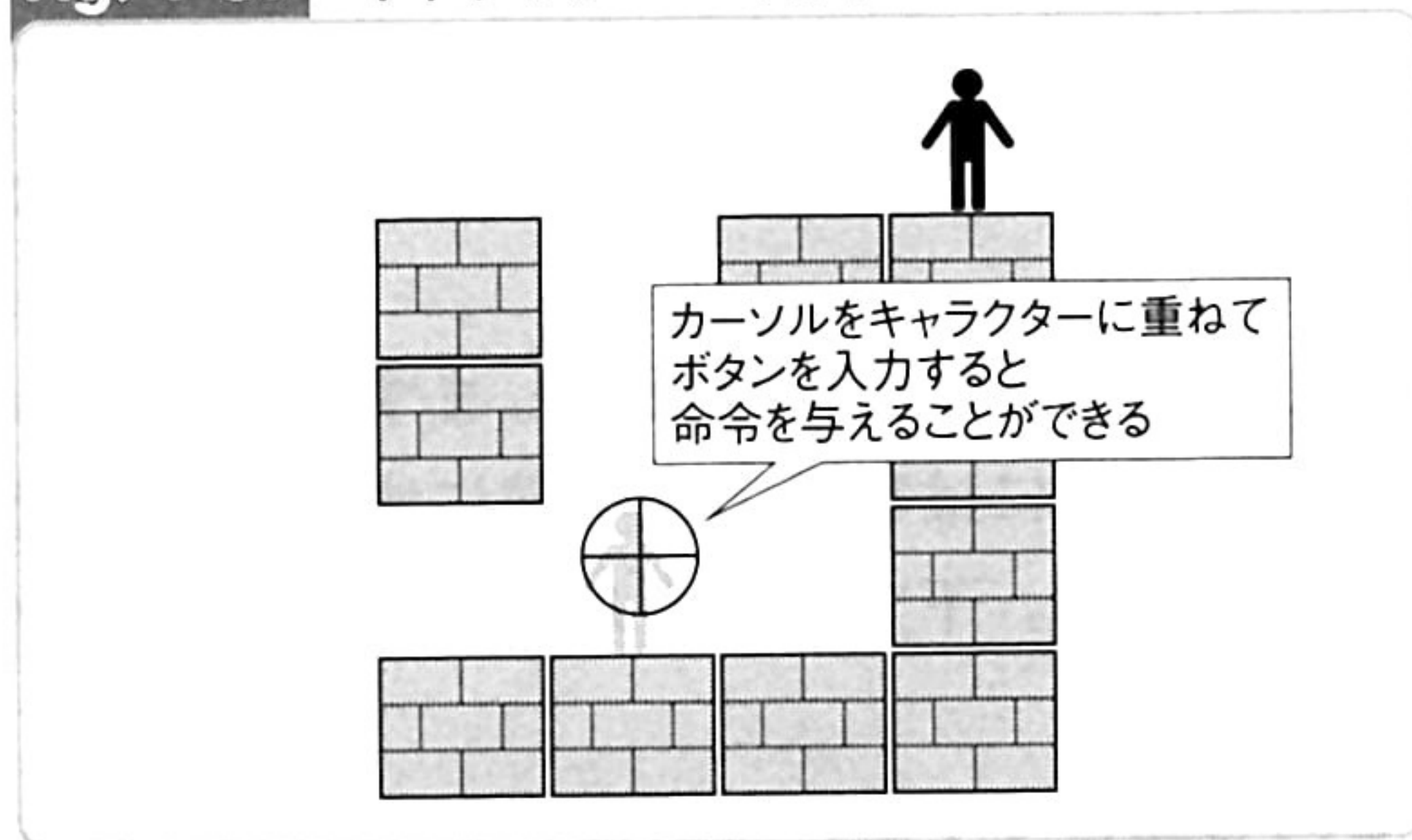
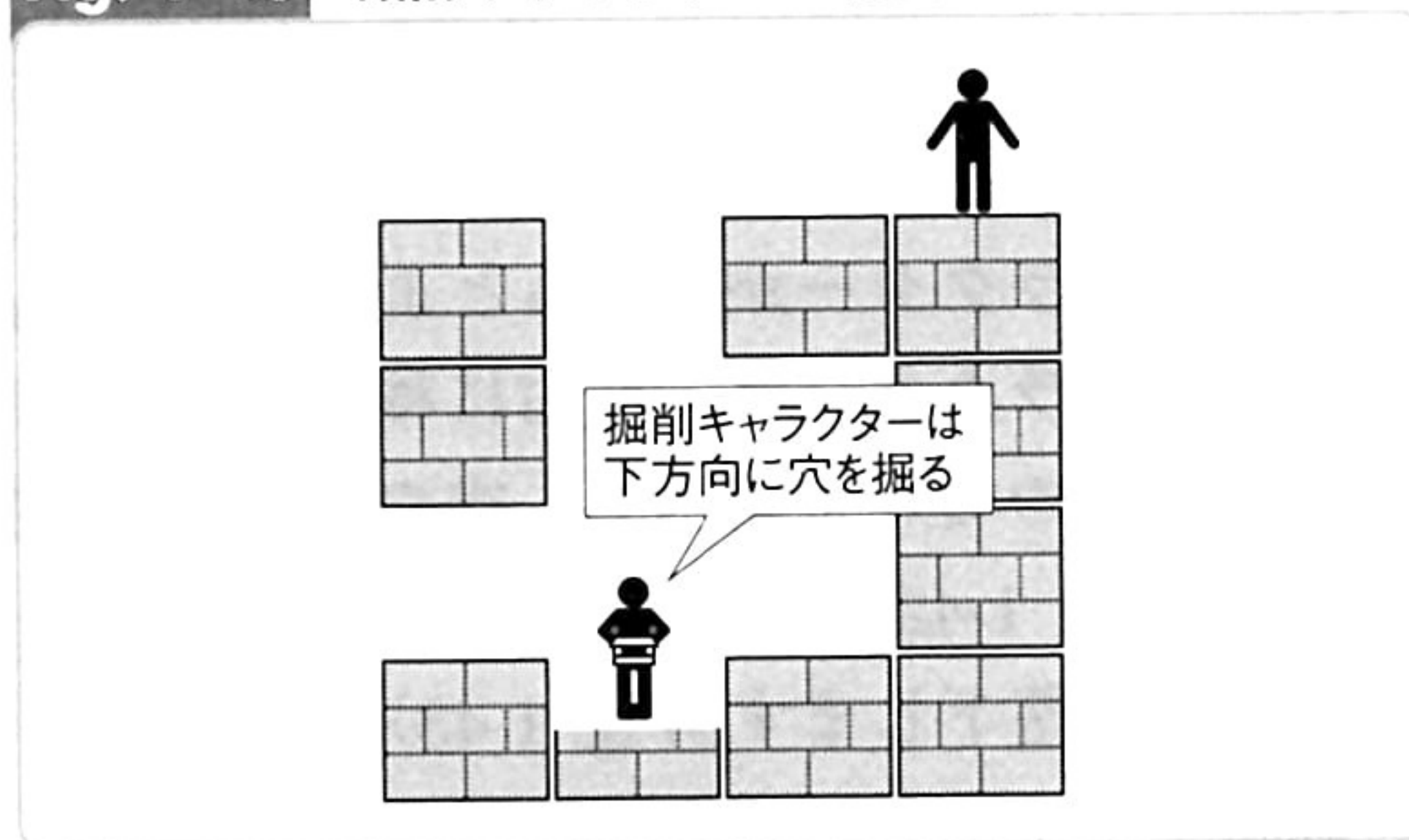
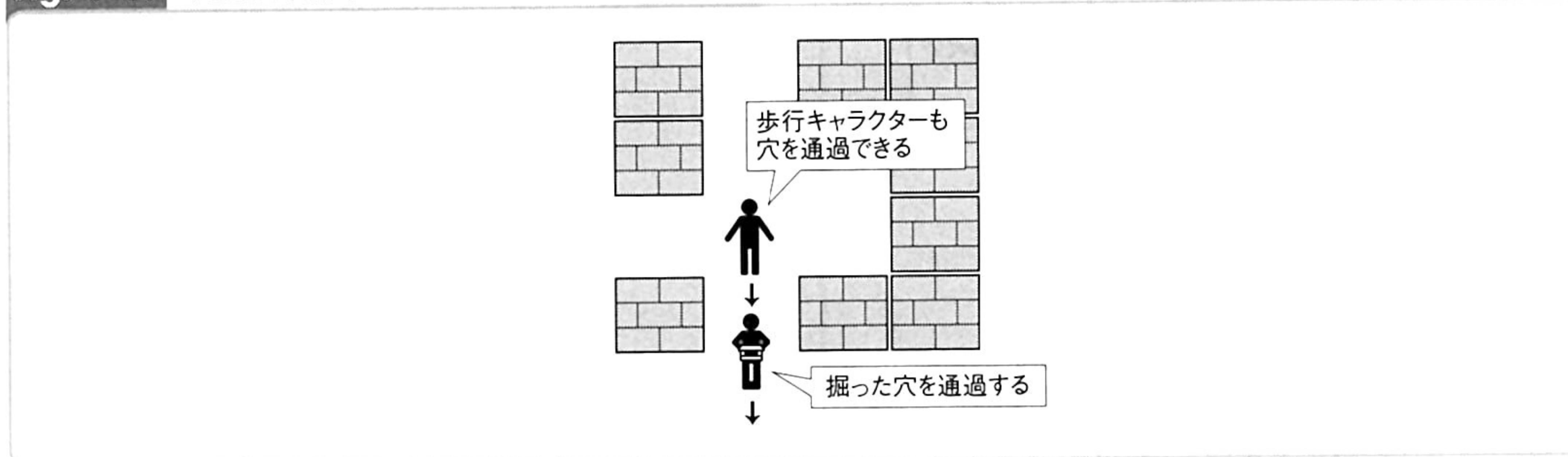



Fig. 1-41 穴を通り抜ける

歩行キャラクターも
穴を通過できる
掘った穴を通過する

導きます。

命令の選択が悪かったり、タイミングが遅かったりすると、キャラクターはさまざまな罠にかかってあっという間に死んでしまいます。命令にはさまざまな種類があり、ステージごとに使える命令の数が決まっているため、効率よく命令を使うための戦略を考える楽しみがあります。

『ピクミン』は『レミングス』によく似たゲームです。このゲームの面白い点は、命令を与えるキャラクターを「笛」を使って決めることです。プレイヤーのキャラクターが笛を吹くと、笛の音が届く範囲のキャラクターが命令を聞きます。笛を吹く長さによって範囲を変えることができるため、操作していて非常に楽しいゲームに仕上がっています。

こういった自律的に動くキャラクターを使ったゲームとしては、パズルゲーム以外にも、多数のRTS（Real-Time Strategy）と呼ばれるジャンルのゲームがあります。RTSの最も代表的な題材は、多数の兵士キャラクターを指揮して、敵軍と戦うというものです。『レミングス』はパズル的な要素が強いゲームですが、『ピクミン』はRTS的な性格も持ったゲームです。

その他『ぐっすんおよよ』にも、自律的に動くキャラクターが登場します。『レミングス』と同じく、キャラクターをゴールに導くことが目的ですが、落ち物パズルゲームのように、ブロックを積み上げてルートを作る点がユニークです。

アルゴリズム

自律的に動くキャラクターは、プレイヤーのレバー入力やボタン入力とは関係なく、自分で状況を判断して行動します。とはいってもそれほど難しいことはなく、ゲームの敵キャラクターを作るときと要領は同じです。賢いキャラクターを作るのは大変ですが、最初はシンプルな動きのキャラクターから作るとよいでしょう。

歩行キャラクターは、左右に移動し、壁に突き当たったら移動方向を反転します。この動きは、移動先のセルを調べて、空の場合には移動し、壁の場合には移動方向を逆にすれば実現できます（Fig. 1-42）。キャラクターの下方向のセルも調べます。下方向のセルが空ならば、キャラクターは落下します（Fig. 1-43）。

掘削キャラクターは歩行キャラクターとほとんど同じ動きですが、下方向のセルが壁のとき

Fig. 1-42 歩行時のセルの状態

#		#	#
#			#
	M →		#
#	#	#	#

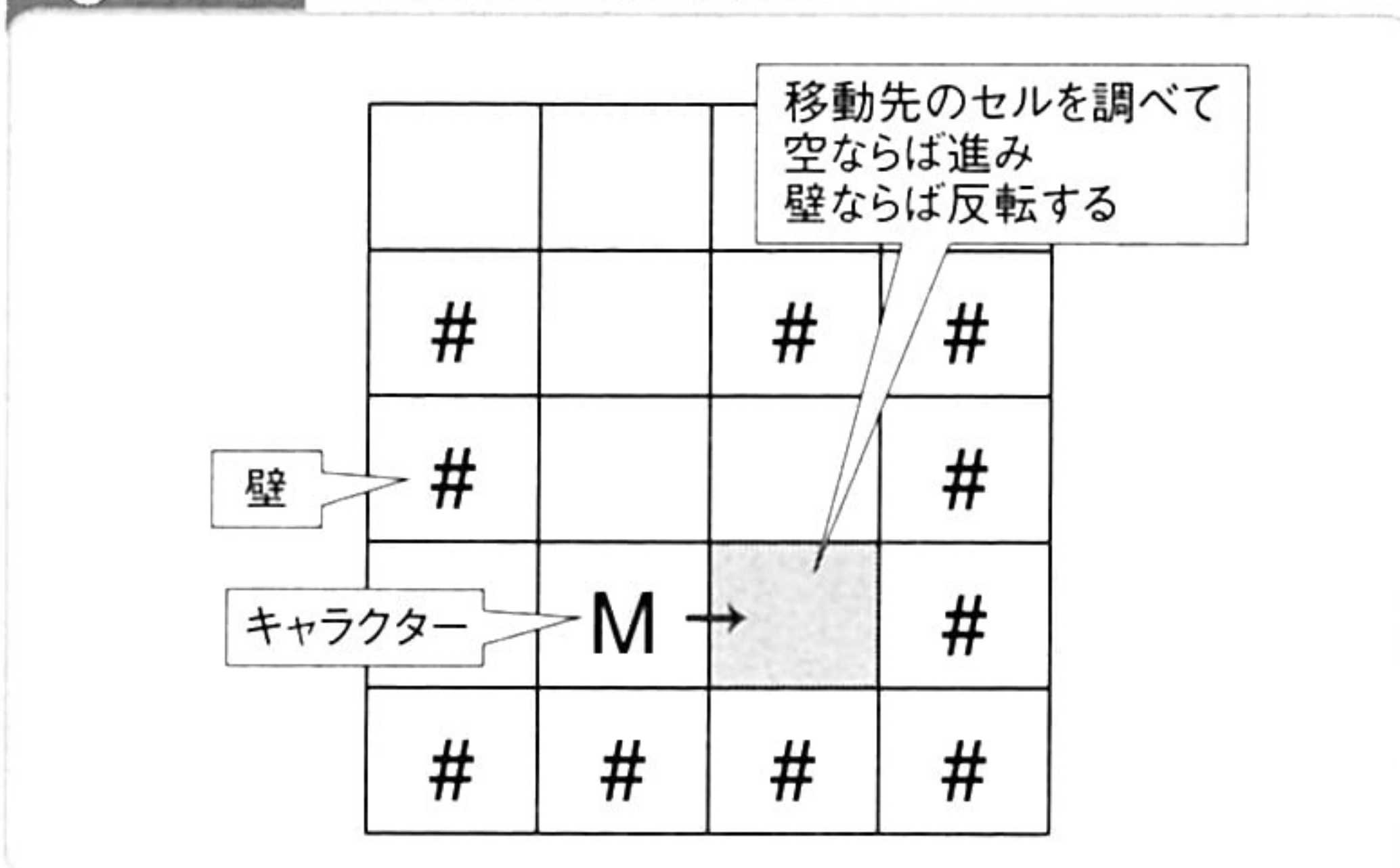
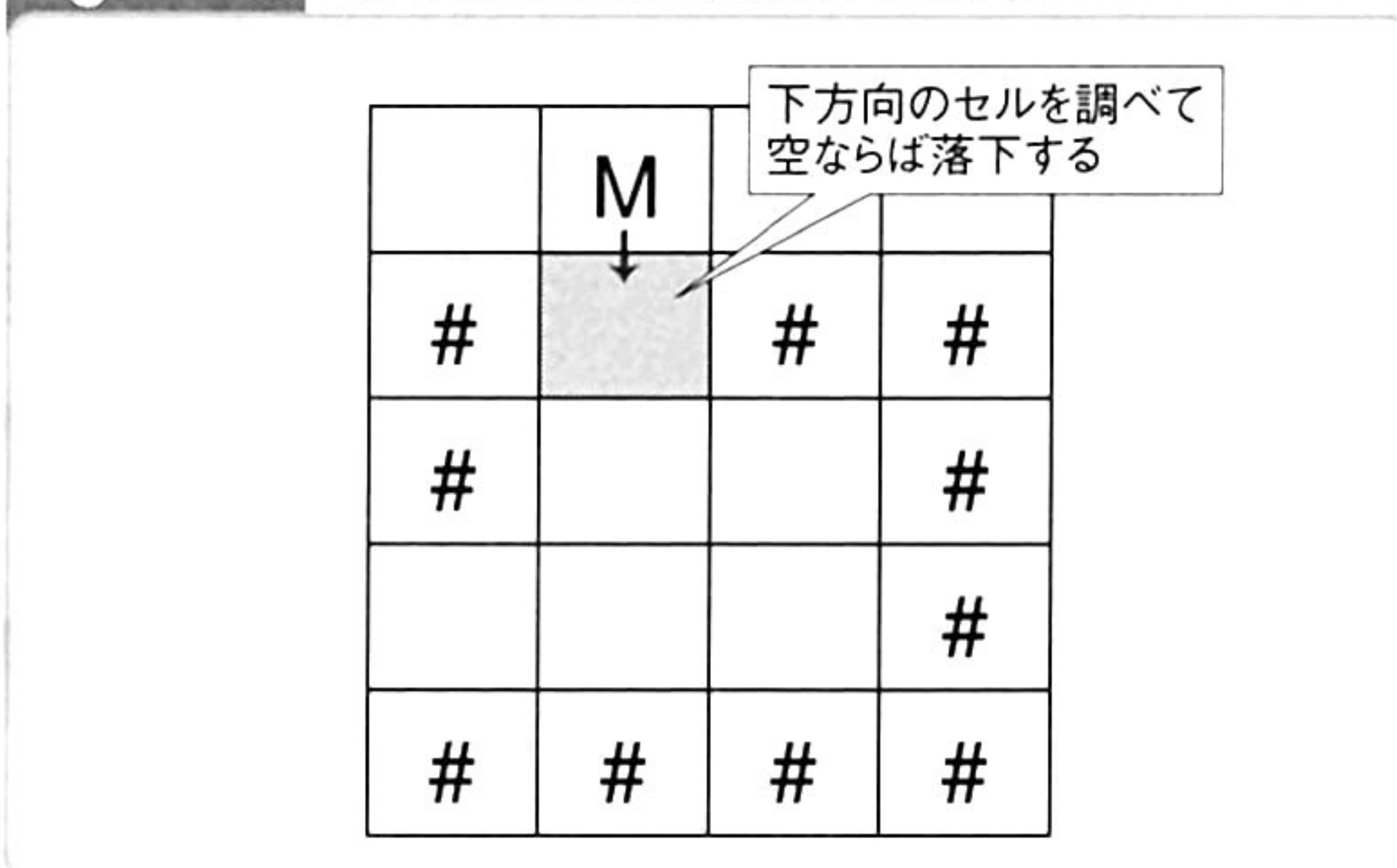
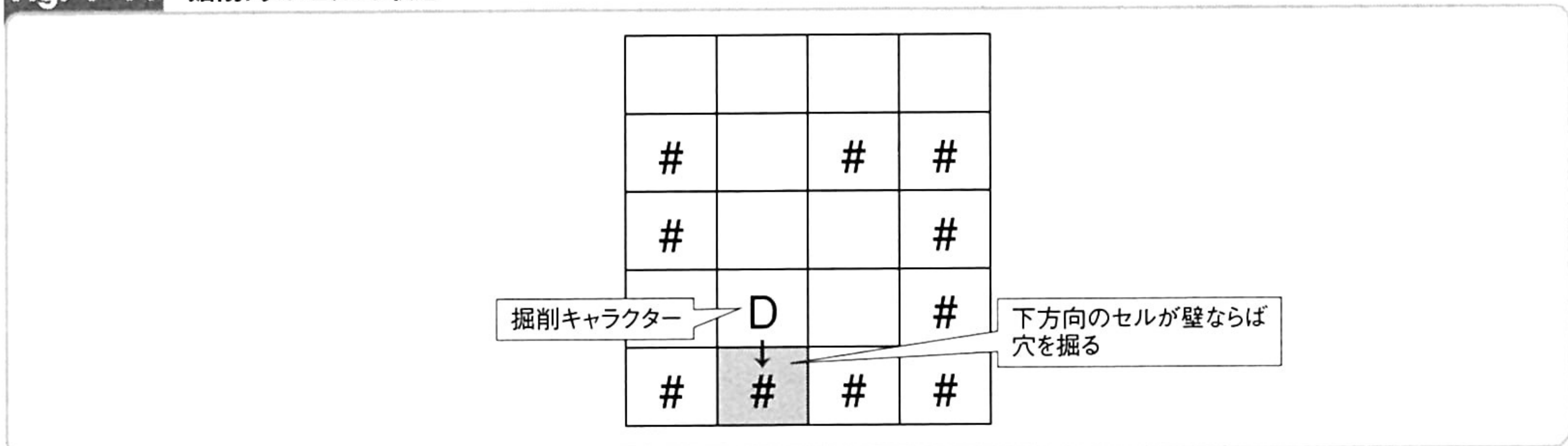



Fig. 1-43 落下時のセルの状態（下方向）

	M ↓		
#		#	#
#			#
			#
#	#	#	#

Fig. 1-44 掘削時のセルの状態

#		#	#
#			#
	D		#
#	#	#	#

だけ動きが違います（Fig. 1-44）。下が壁のとき、掘削キャラクターは穴を掘ります。穴になったセルは壁から空に変化させて、キャラクターが通過できるようにします。なお、図では掘削キャラクターを文字「D」（Digger）で示しました。

プログラム

List 1-5は自律的に動くキャラクターのプログラムです。キャラクターとカーソルの移動処理、およびキャラクターに穴掘りの命令を出すときの処理を抜粋しました。

キャラクターには静止・落下・掘削・歩行という4種類の状態があります。静止状態で下方向や左右方向のセルを調べ、条件に応じて他の状態に移行します。

なお、このプログラムではセルにキャラクターを書き込みません。同一のセルを複数のキャラクターが重なって通過することがあるため、セルに書き込むと不都合が出るためです。

また、画面の最下部には掘れない壁を配置しています。これは掘削キャラクターが画面下端の壁を掘り抜いて、画面外に出てしまうことを防ぐためです。

カーソルの移動処理では、カーソルとすべてのキャラクターとの間で当たり判定処理を行います。カーソルに接触しているキャラクターが存在するときには、カーソルの形を変えます。このときにボタン入力があれば、接触しているキャラクターに対して命令を発行し、掘削キャラクターに変化させます。

List 1-5 自律的に動くキャラクター（CSelfDirectiveCharacterManクラス、CSelfDirectiveCharacterCursorクラス）

```
// キャラクターを命令を出すときに呼び出す処理
void CSelfDirectiveCharacterMan::Dig() {

    // 掘削キャラクターのフラグを設定する
    Digger=true;

    // グラフィックを掘るキャラクターに変更する
    Texture=Game->Texture[TEX_DRILLER];
}
```

```
// キャラクターの移動処理
bool CSelfDirectiveCharacterMan::Move(const CInputState* is) {

    // 静止状態の処理
    if (State==0) {

        // 下方向のセルが空の場合には、落下状態へ移行する
        if (Cell->Get(CX, CY+1)==' ') {

            // セル座標の更新
            CY+=VY;

            // タイマーと状態の設定
            Time=20;
            State=1;
        } else

        // 掘削キャラクターのフラグが設定されていて、
        // かつ下方向のセルが壁の場合には、掘削状態へ移行する
        if (Digger && Cell->Get(CX, CY+1)=='#') {

            // セル座標の更新
            CY+=VY;

            // タイマーと状態の設定
            Time=40;
            State=2;
        } else

        // 左右方向のセルが空の場合には、歩行状態に移行する
        if (Cell->Get(CX+VX, CY)==' ') {

            // セル座標の更新
            CX+=VX;

            // タイマーと状態の設定
            Time=20;
            State=3;
        } else {

            // 左右方向のセルが空ではないときには、
            // 移動方向を逆にする
            VX=-VX;
        }
    }

    // 落下状態の処理
    if (State==1) {
```

```
        // タイマーの更新
        Time--;

        // 描画座標の更新
        Y=CY-VY*Time*0.05f;

        // タイマーが0になったら静止状態へ移行する
        if (Time==0) {
            State=0;
        }
    }

    // 掘削状態の処理
    if (State==2) {

        // タイマーの更新
        Time--;

        // 描画座標の更新
        Y=CY-VY*Time*0.025f;

        // タイマーが0になったら、
        // 掘った床のセルを空にして、
        // 静止状態へ移行する
        if (Time==0) {
            Cell->Set(CX, CY, ' ');
            State=0;
        }
    }

    // 歩行状態の処理
    if (State==3) {

        // タイマーの更新
        Time--;

        // 描画座標の更新
        X=CX-VX*Time*0.05f;

        // タイマーが0になったら静止状態へ移行する
        if (Time==0) {
            State=0;
        }
    }
    return true;
}

// カーソルの移動処理
bool CSelfDirectiveCharacterCursor::Move(const CInputState* is) {
```

```
	// カーソルのスピード、当たり判定の大きさ
	float speed=0.1f, hit=0.5f;

	// レバー入力に応じて、カーソルの座標を更新する
	if (is->Left && X>0) X-=speed;
	if (is->Right && X<MAX_X-1) X+=speed;
	if (is->Up && Y>0) Y-=speed;
	if (is->Down && Y<MAX_Y-1) Y+=speed;

	// すべてのキャラクターについて、
	// カーソルとの当たり判定処理を行う
	CSelfDirectiveCharacterMan* man=NULL;
	for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
		CMover* mover=(CMover*)i.Next();

		// カーソルに接触しているキャラクターを見つけたら、
		// ループを抜け出す
		if (
			mover!=this &&
			mover->X<X+hit && X<mover->X+hit &&
			mover->Y<Y+hit && Y<mover->Y+hit
		) {
			man=(CSelfDirectiveCharacterMan*)mover;
			break;
		}
	}

	// カーソルに接触しているキャラクターがいる場合の処理
	if (man) {

		// カーソルを円形にする
		Texture=Game->Texture[TEX_PILE];

		// ボタン入力があれば、
		// カーソルに接触しているキャラクターに命令を出す
		if (is->Button[0]) {
			man->Dig();
		}
	} else {

		// カーソルに接触しているキャラクターがいない場合には、
		// カーソルを十字型にする
		Texture=Game->Texture[TEX_SHURIKEN];
	}
	return true;
}
```

SAMPLE

「SELF DIRECTIVE CHARACTER」は「自律的に動くキャラクター」のサンプルです。レバーの上下左右（カーソルキーの上下左右）でカーソルが移動します。

画面では多数のキャラクターが自動的に動き回っています。キャラクターは左右に動き、壁にぶつかると方向転換します。また、下に壁がないときには落下します。

キャラクターにカーソルを合わせて、ボタン0（Zキー）を押すと、キャラクターに命令を与えることができます。命令を受けたキャラクターは、自分の下にある壁を掘って、穴を開けるようになります。命令を受けたキャラクターも、受けていないキャラクターも、開いた穴を通り抜けることができます。

SELF DIRECTIVE CHARACTER → p. 386

遅れて追随するカーソル

レバー入力でカーソルを動かすと、別のカーソルが遅れて追いかけてくるアクションです。遅延を上手に利用することによって、複数のカーソルにまったく別々の場所をポイントさせることができます。

ここではレバー入力で動かせるカーソルを「先頭カーソル」、先頭カーソルに追随するカーソルを「追随カーソル」、末尾の追随カーソルを「末尾カーソル」と呼ぶことにしましょう（Fig. 1-45）。先頭カーソルが移動した後には、複数の追随カーソルによる軌跡が残ります。図では先頭カーソルと末尾カーソルを黒色で、末尾以外の追随カーソルを白色で示しました。

先頭カーソルと追随カーソルを使った操作の例として、Fig. 1-46のような場合を考えてみましょう。壁に囲まれた左右2つの区画に、それぞれボールが入っています。これから先頭カーソルと追随カーソルを使って、2つの区画のなかにあるボールを入れ替えてみます。

追随カーソルは少し遅れて先頭カーソルを追いかけるので、各カーソルに別々の区画をポイントさせることができます。先頭カーソルと末尾の追随カーソルが異なる区画を指している瞬間を狙って、ボタンを押します。タイミングよくボタンを入力すると、2つの区画の中身を入れ替えることができます（Fig. 1-47）。

Fig. 1-45 先頭カーソルと軌跡

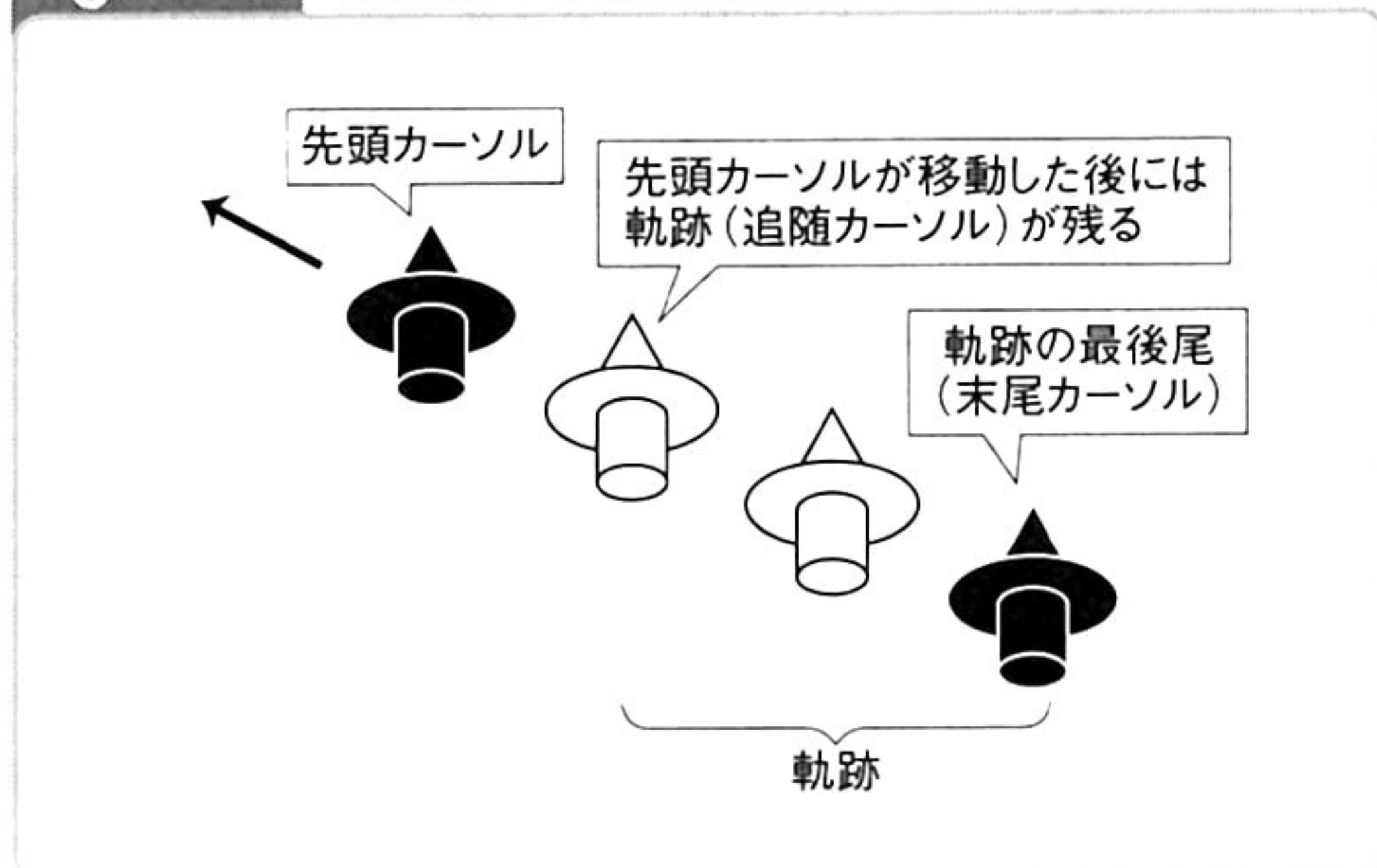



Fig. 1-46 壁に囲まれた2つの区画

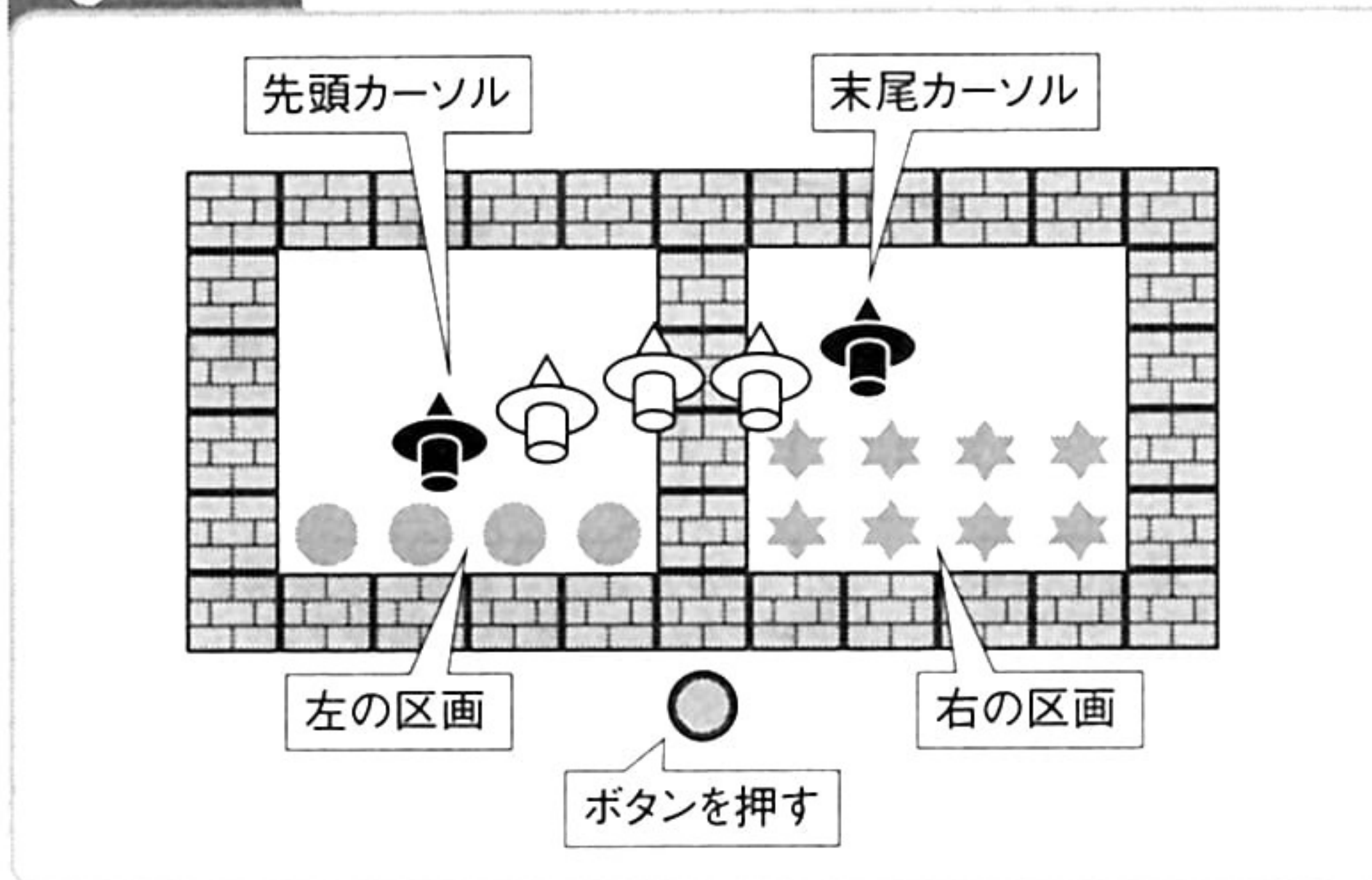



Fig. 1-47 カーソルを使って区画の中身を入れ替える

先頭カーソルが指していた区画と
末尾カーソルが指していた区画の
中身が入れ替わる

遅れて追随するカーソルを使ったゲームには『ロットロット』があります。このゲームではステージが複数の区画に分かれており、区画間の壁は時間とともに開閉します。画面上方からは多数のボールが降ってきて、壁が開いている区画にどんどんたまっていきます。

ほうっておくと、壁の開閉にともなってボールはどんどん下に落ちてしまいます。ボールが特定の場所に落ちるとミスになってしまうため、その場所に落ちないように、カーソルを使ってボールを安全な場所に誘導します。

先頭カーソルと末尾カーソルが別の区画を指しているときに、タイミングよくボタンを押すと、区画内のボールを入れ替えることができます。上手にボールを入れ替えて、ミスをせずにスコアを稼ぎ続けることがゲームの目的です。追随カーソルによる独特の操作感と、ボールや壁のトリッキーな動きが楽しめる、ユニークなゲームです。なお、ボールの動きについては、「転がる大量のボール」(→p. 352) を参照してください。

アルゴリズム

遅れて追随するカーソルを実現するには、先頭カーソルが通った座標を、フレーム (例えば1/60秒) ごとに配列に保存しておきます (Fig. 1-48)。そして、数フレームおきに配列から座標を取り出して、追随カーソルや末尾カーソルの座標にします (Fig. 1-49)。配列を長くすればす

Fig. 1-48 先頭カーソルの座標を配列に保存する

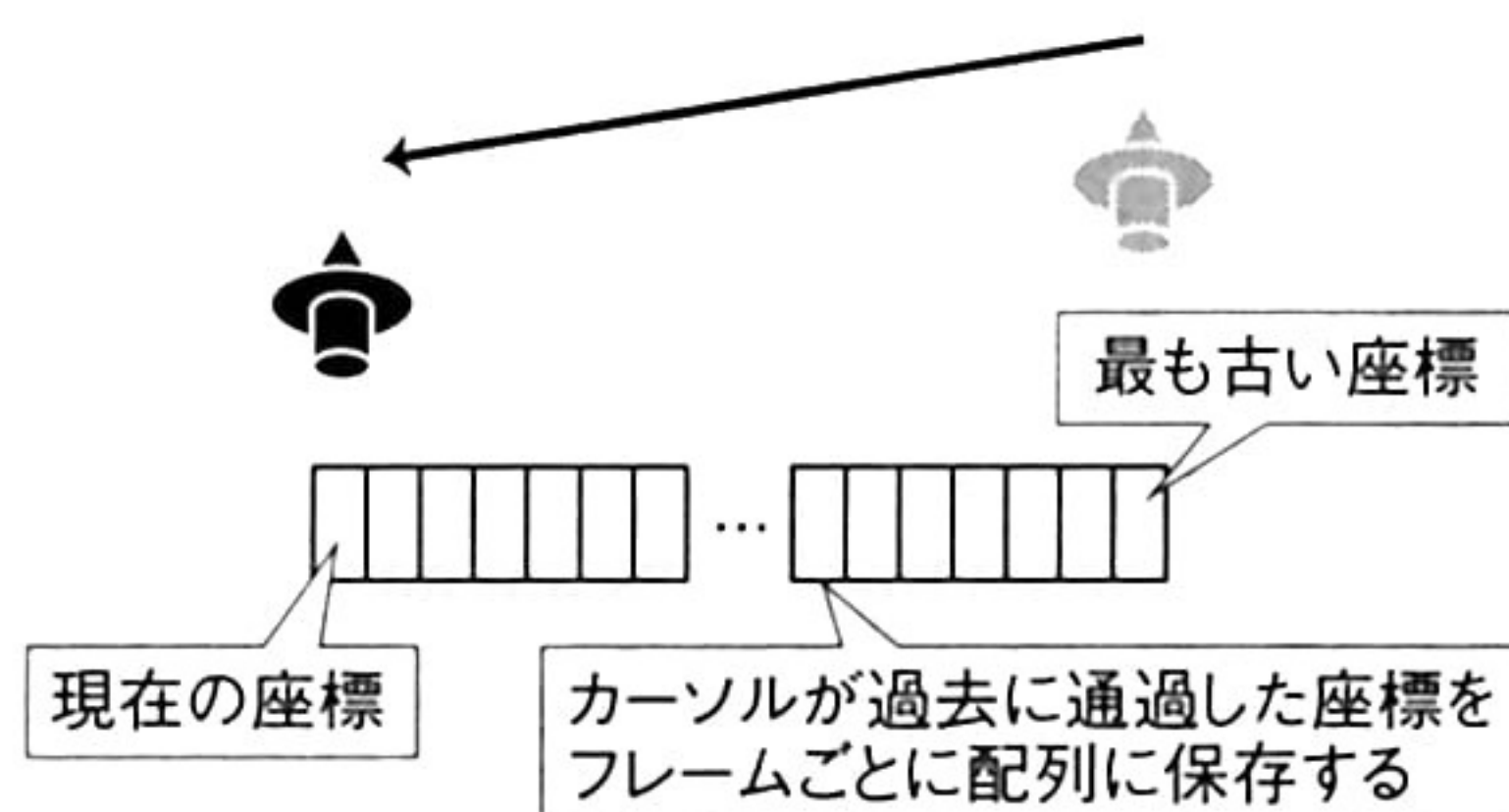



Fig. 1-49 配列から座標を取り出す

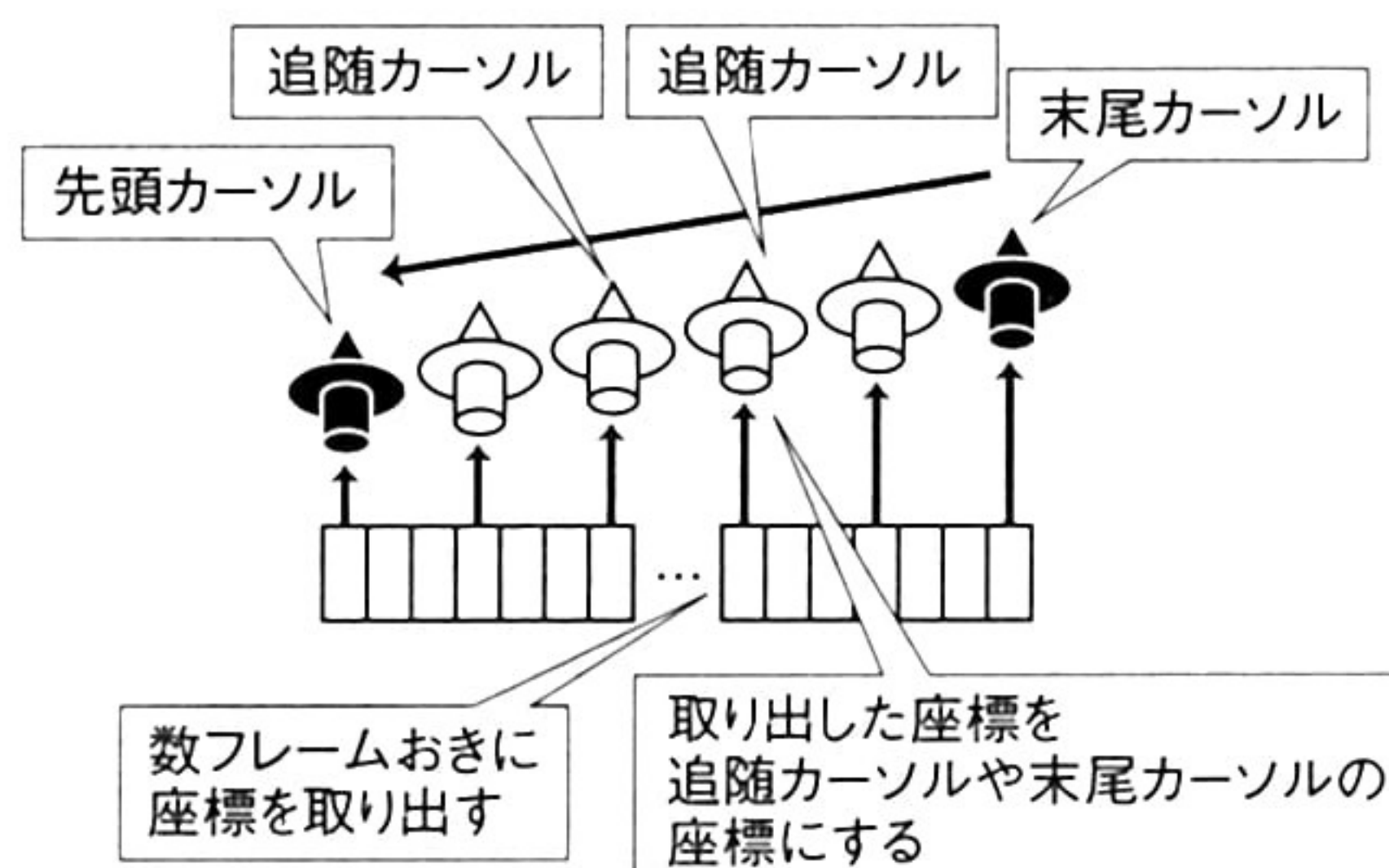



Fig. 1-50 最新の座標を配列の先頭に格納する方法

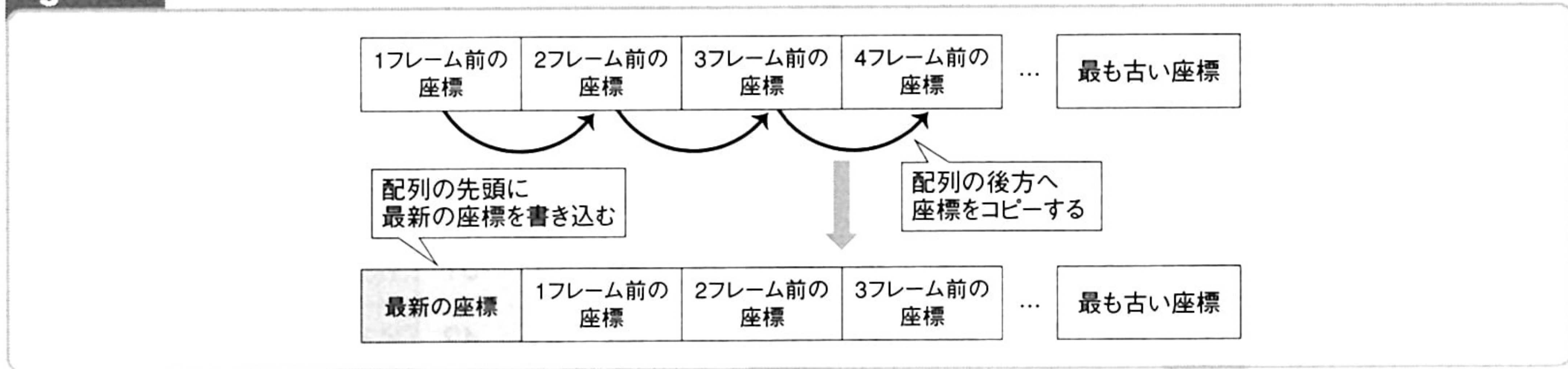
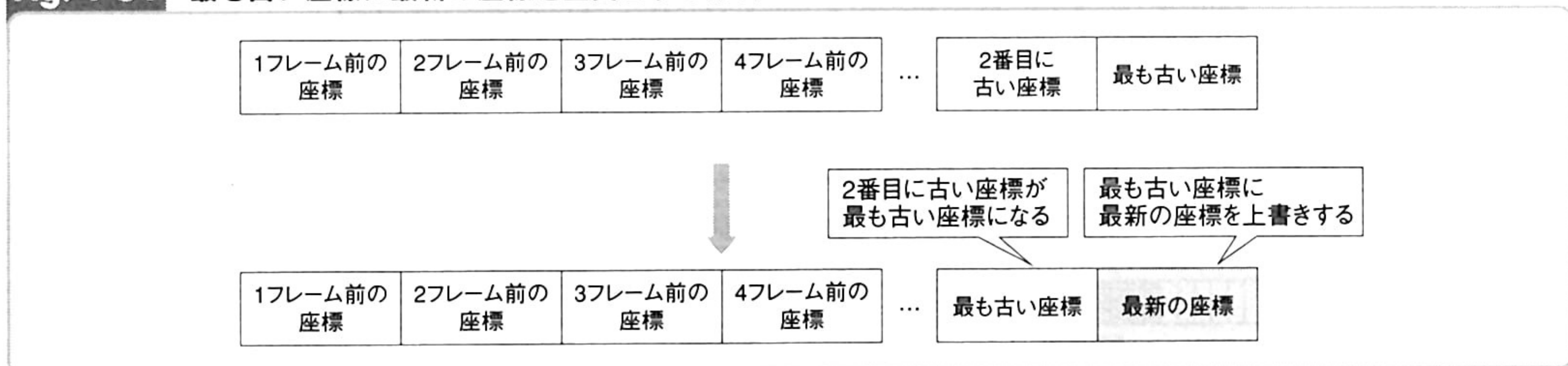



Fig. 1-51 最も古い座標に最新の座標を上書きする方法

るほど、長い軌跡を作ることができます。

配列に座標を保存するときには、2つの方法があります。1つは最新の座標を常に配列の先頭へ格納する方法です（Fig. 1-50）。カーソルが移動するたびに、古い座標を配列の後方へコピーしてから、新しい座標を配列の先頭に書き込みます。

もう1つの方法は、最も古い座標に最新の座標を上書きする方法です（Fig. 1-51）。次のフレームでは、2番目に古い座標が最も古い座標になるので、ここに最新の座標を上書きします。つまり、最新の座標を書き込む位置が、だんだん配列の前方へ移動していきます。

前者の利点は、最新の座標が常に配列の先頭にあるため、処理が簡単になることです。ただし、配列をコピーする処理が必要になるという欠点があります。後者の利点は、配列のコピーが不要になることです。ただし、最新の座標を書き込む位置が変化するため、処理は多少複雑になります。サンプルでは、たかだか要素数が100個程度の配列のコピーは重い処理ではないことと、プログラムがシンプルになることを考慮して、前者の方法を採用しています。

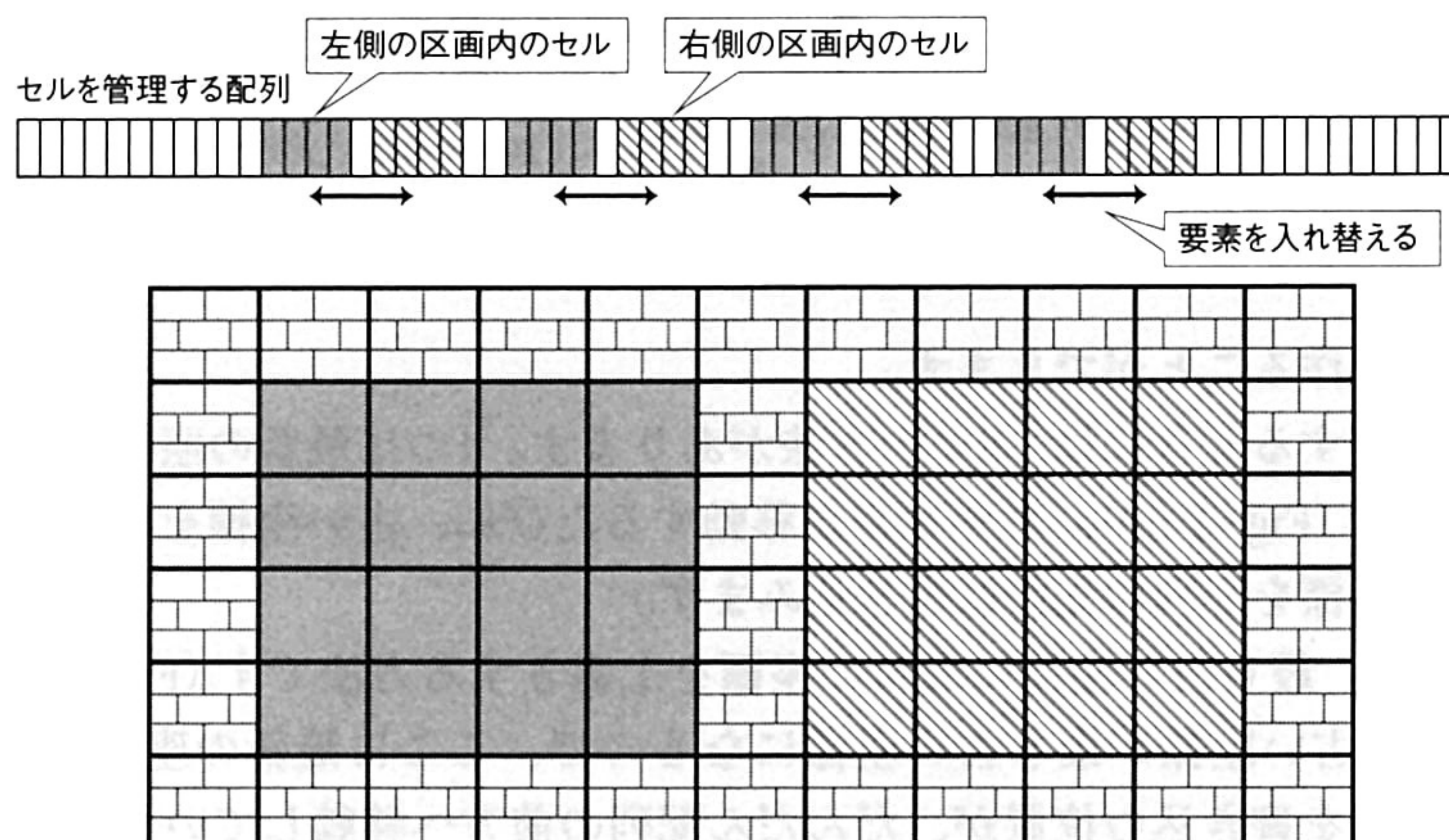
ボールの入れ替え

カーソルを使ったボールの入れ替えについては、区画内のセルを入れ替えることによって実現します。配列を使ってセルを管理した場合の処理について、少し具体的に解説しましょう。

セルは1次元配列でも2次元配列でもかまいませんが、ここでは1次元配列で管理することにします（Fig. 1-52）。セルの番号は配列の添字（配列内で各セルが何番目の要素に相当するのか）を示します。図では左上から右下へ向かって番号を割り当てました。

区画内のセルを入れ替えるには、配列上で区画内に対応する要素を入れ替えます（Fig. 1-53）。

Fig. 1-52 セルを配列で管理する

Fig. 1-53 最新の座標を配列の先頭に格納する方法

図ではそれぞれの区画に対応する要素を、色分けで示しました。これで、セルの中身を入れ替えることができます。

プログラム

List 1-6は遅れて追随するカーソルのプログラムです。先頭カーソルと追随カーソルの移動処理を掲載しました。

先頭カーソルの移動処理では、レバー入力に応じてカーソルを動かします。カーソルの座標はフレームごとに配列に記録しておきます。ボタンを押したときには、先頭カーソルが指す区画と、末尾カーソルが指す区画との間で、セルを入れ替えます。

追随カーソルの移動処理では、先頭カーソルの座標を記録した配列から座標を取り出し、追随カーソルの座標にします。複数の追随カーソルを生成し、配列から座標を取り出す位置をそ

れぞれ変えることによって、各々のカーソルが少しずつ遅れながら先頭カーソルを追いかけます。これで、いくつもの追随カーソルが先頭カーソルの軌跡を描くような効果が得られます。

List 1-6　遅れて追随するカーソル(CFollowingCursorHeadCursorクラス、CFollowingCursorTailCursorクラス)

```
// カーソルが過去に通った座標を保存するための配列
#define FOLLOWING_CURSOR_LENGTH 101
float
    FollowingCursorX[FOLLOWING_CURSOR_LENGTH],
    FollowingCursorY[FOLLOWING_CURSOR_LENGTH];

// 先頭カーソルの移動処理
bool CFollowingCursorHeadCursor::Move(const CInputState* is) {

    // カーソルの移動スピード
    float speed=0.1f;

    // レバー入力に応じてカーソルの描画座標を変化させる
    if (is->Left && X>0) X-=speed;
    if (is->Right && X<MAX_X-1) X+=speed;
    if (is->Up && Y>0) Y-=speed;
    if (is->Down && Y<MAX_Y-1) Y+=speed;

    // ボタンを入力したときの処理
    // 先頭カーソルが指す区画と
    // 末尾の追随カーソルが指す区画の間で
    // セルを入れ替える
    float *fx=FollowingCursorX, *fy=FollowingCursorY;
    if (!PrevButton && is->Button[0]) {

        // カーソルが指す区画のセル座標を計算する
        int
            x=((int)X/5)*5,
            y=(((int)Y-1)/5)*5+1,
            tx=((int)fx[FOLLOWING_CURSOR_LENGTH-1]/5)*5,
            ty=(((int)fy[FOLLOWING_CURSOR_LENGTH-1]-1)/5)*5+1;

        // 区画内のセルを入れ替える
        for (int i=1; i<=4; i++) {
            for (int j=1; j<=4; j++) {
                Cell->Swap(x+i, y+j, tx+i, ty+j);
            }
        }
    }

    // ボタンを押しっぱなしにしたときに
    // 入れ替えが連続して行われないために
    // 前回のボタン入力を記録する
    PrevButton=is->Button[0];

    // カーソルの座標を記録するために
```

```
    //古い座標を配列の後方へずらす
    for (int i=FOLLOWING_CURSOR_LENGTH-1; i>0; i--) {
        fx[i]=fx[i-1];
        fy[i]=fy[i-1];
    }

    //配列の先頭に最新の座標を記録する
    fx[0]=X;
    fy[0]=Y;

    return true;
}

//追随カーソルの移動処理
bool CFollowingCursorTailCursor::Move(const CInputState* is) {

    //先頭カーソルの古い座標を配列から取り出して
    //追随カーソルの座標にする
    float *fx=FollowingCursorX, *fy=FollowingCursorY;
    X=fx[Delay];
    Y=fy[Delay];

    return true;
}
```

SAMPLE

「FOLLOWING CURSOR」は「遅れて追随するカーソル」のサンプルです。レバーの上下左右(カーソルキーの上下左右)でカーソルが移動します。

先頭のカーソルの後を少し遅れて、複数のカーソルが追いかけます。追随するカーソルは白色で、先頭と末尾のカーソルは黒色です。

画面は6つの区画に分けられていて、各区画にボールが配置されています。ボタン0(Zキー)を押すと、先頭のカーソルが指す区画のボールと、末尾のカーソルが指す区画のボールが入れ替わります。

FOLLOWING CURSOR → p. 386

まとめ

本章ではいろいろな「動かす」アクションを集めてみました。例えば「荷物を押して動かす」というだけでも、さまざまな動きのバリエーションがあり、動きが違えばゲーム性も変わります。ゲームのルールや、ステージの作り方もまったく変わってきます。

というわけで、「動きがゲームを作る！」というのが本章のまとめです。

Stage
02

落とす

Drop

「ものを落とす」というアクションを採用したゲームは数多くあります。こういったゲームは「落ち物パズルゲーム」と呼ばれていて、一大ジャンルを築き上げています。「落とす」という動きは共通であるものの、細かいルールはゲームによって違い、ルールの違いによってゲーム性は意外なほど変わってきます。

ブロックを落とす

ブロックを落として積み上げるアクションです。ブロックを移動させたり、回転させたりしながら、隙間なく積み上げることがゲームの目的です。

パズルのピースのように、ブロックはさまざまな形をしています（Fig. 2-1）。ブロックは、4つの小さな矩形を組み合わせて構成されています。ゲームによっては、3つの矩形を組み合わせて1つのブロックにしたり、5つ以上の矩形を組み合わせていたりするのものもあります。

ゲームが始まると、いろいろな形状のブロックのなかから、ランダムに1つのブロックが出現します。ブロックは画面上方に出現して、時間とともに画面下方へ落ちていきます（Fig. 2-2）。

初級のステージでは、ブロックはゆっくりと落ちてきます。しかしステージが進むと落下スピードがしだいに上がり、上級のステージでは矢のような速さでブロックが降り注ぎます。スピードに負けずに、ブロックをさばき続けられるかどうかが、プレイヤーの腕の見せ所です。

レバーを下に入力すると、ブロックの落下スピードを上げることもできます（Fig. 2-3）。特に初級ステージでは、ブロックの落下が遅いので、積極的に落下スピードを上げるプレイが一般的です。画面下端に着地すると、ブロックは落下地点に固定されます（Fig. 2-4）。また、積み上げられた他のブロックの上に着地したときにも、ブロックは固定されます（Fig. 2-5）。い

Fig. 2-1 さまざまな形状のブロック

4つの小さな矩形ブロックを
組み合わせて
さまざまな形状のブロックを作る

Fig. 2-2 落下するブロック

画面上方に
ブロックが出現する
（種類はランダムに決まる）
時間とともに
少しずつ落下する

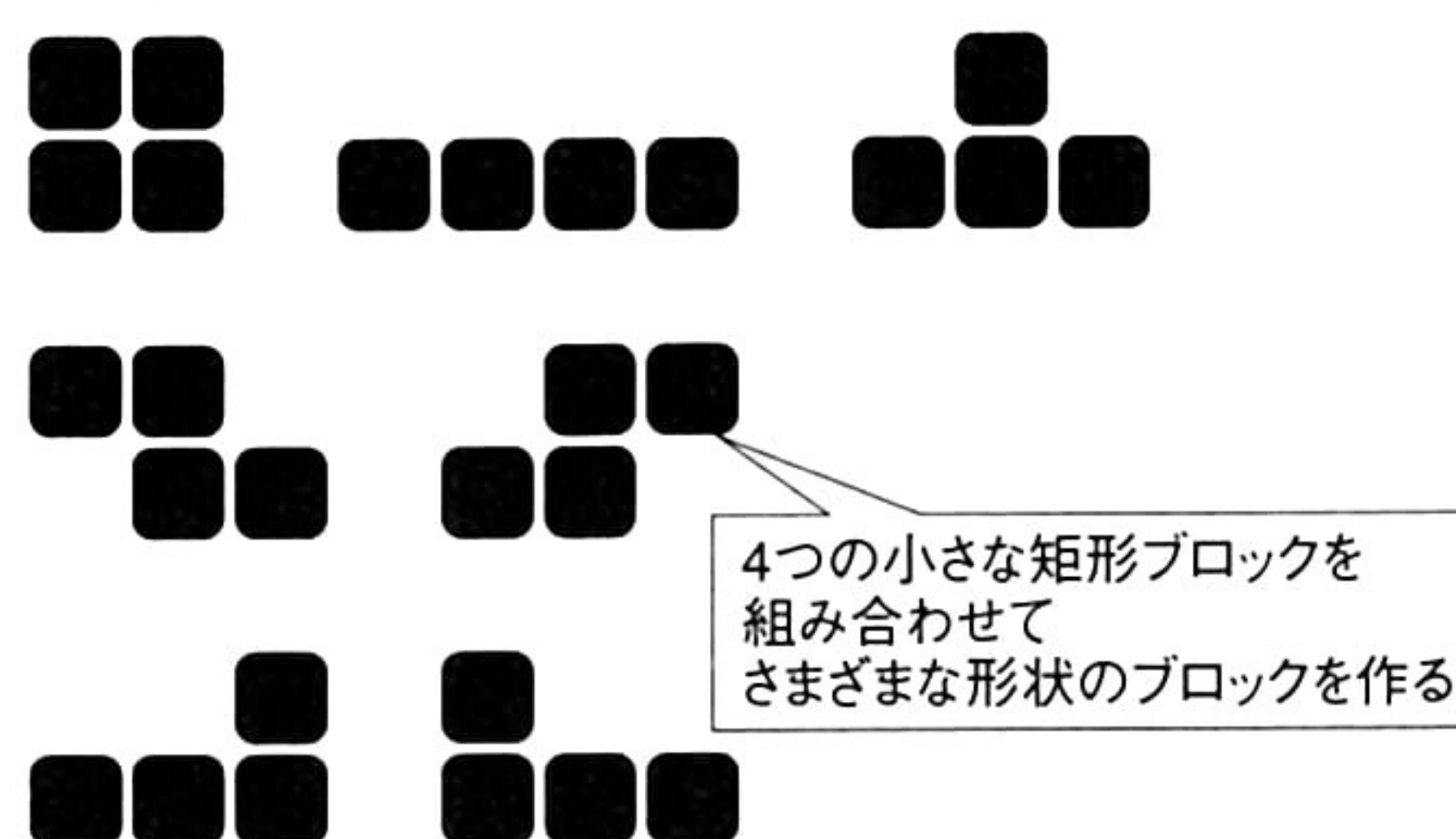
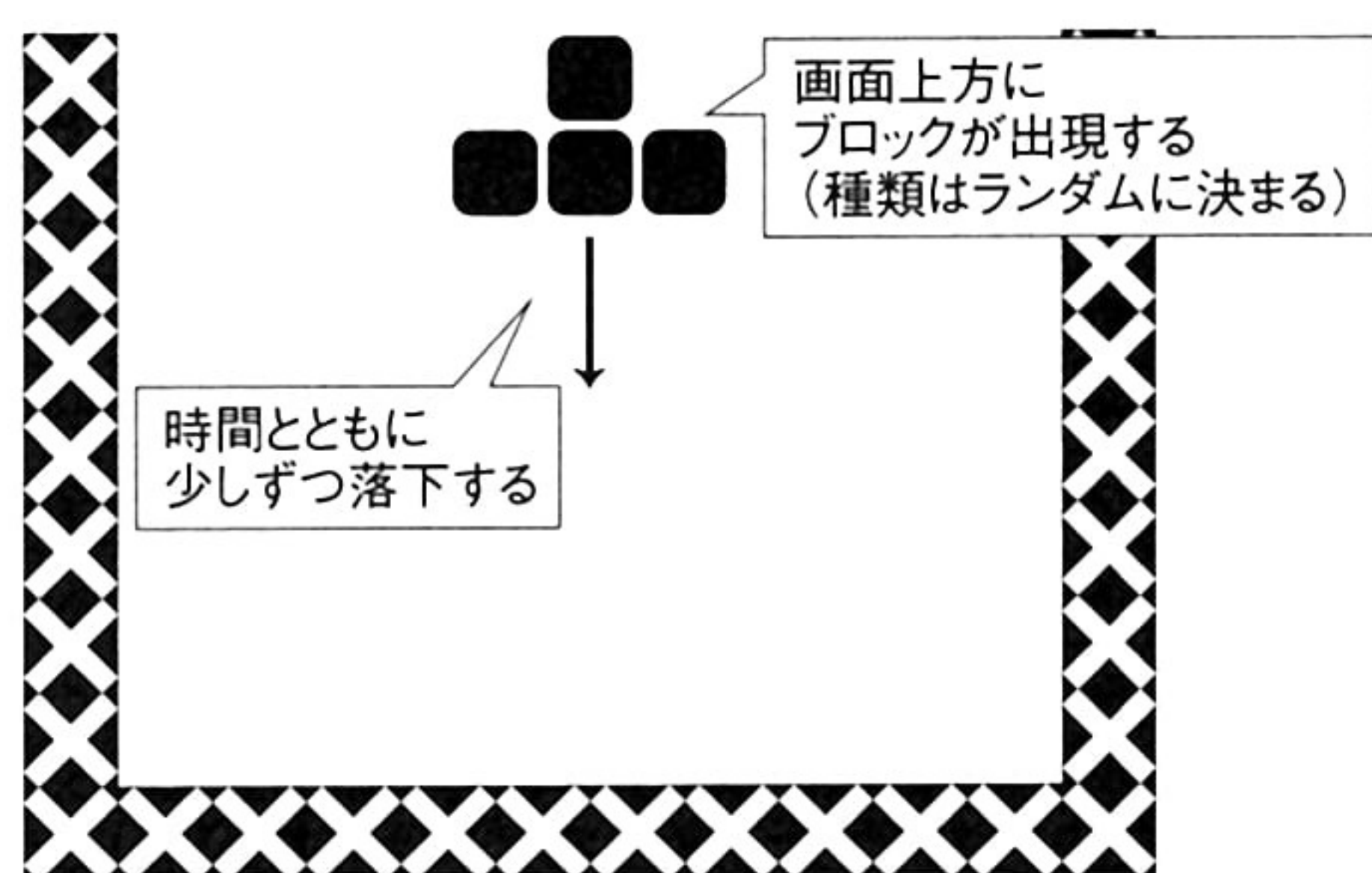



Fig. 2-3 レバー入力で落下スピードを上げる

レバーを下に入力
落下スピードが
上がる

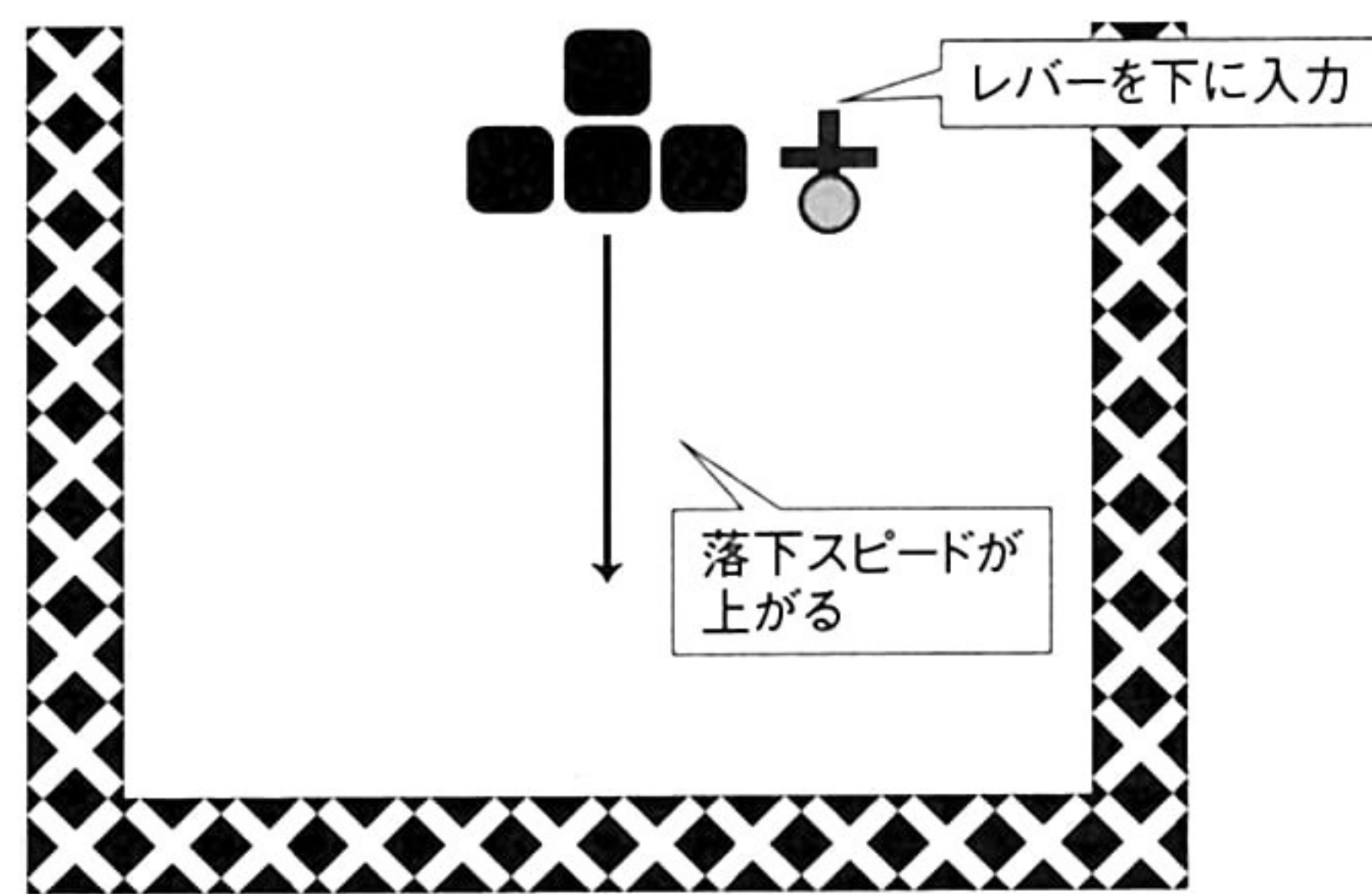



Fig. 2-4　ブロックが画面下端に着地する

画面下端に達すると
ブロックは着地する

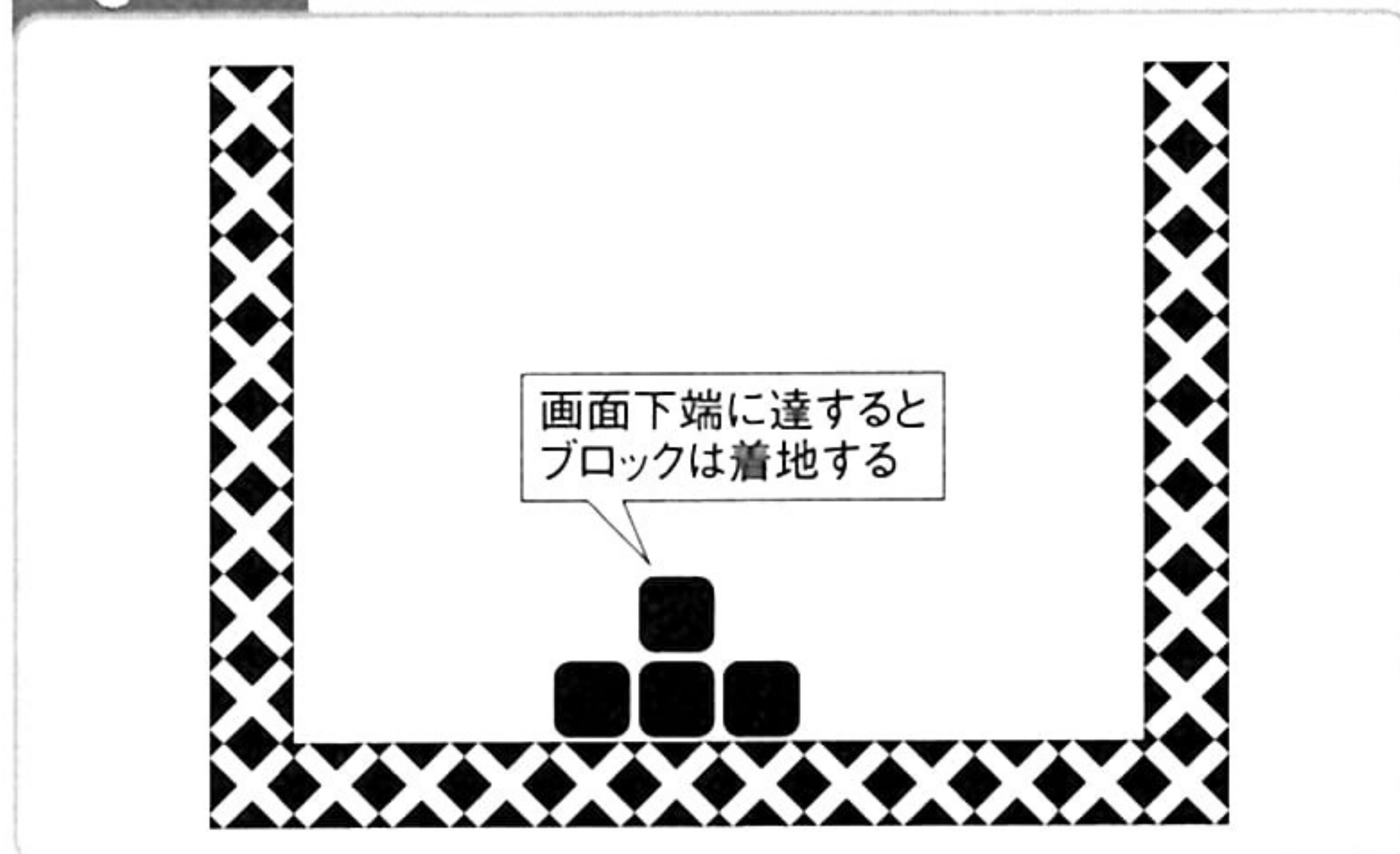
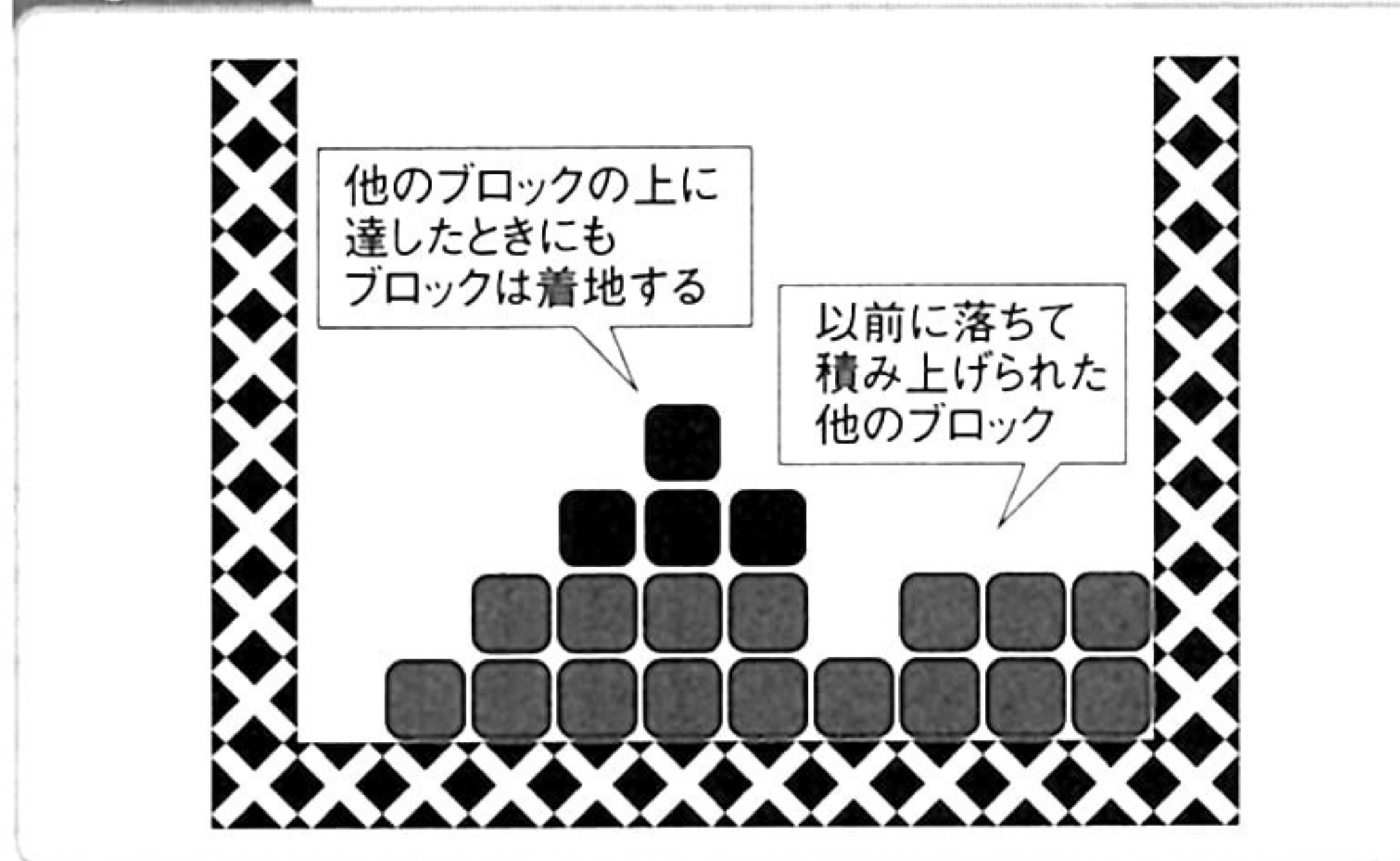



Fig. 2-5　ブロックが他のブロック上に着地する

他のブロックの上に
達したときにも
ブロックは着地する

以前に落ちて
積み上げられた
他のブロック

ずれの場合も、画面上端には次のブロックが出現します。

ブロックを落として積み上げるゲームとしては、『テトリス』が有名です。このゲームでは、画面上方からいろいろな形状のブロックが落ちてきます。ブロックを横1段に隙間なく詰めると、その段を消すことができます。次々と落ちてくるブロックを上手にさばいて、消していくことがゲームの目的です。『テトリス』に類似したゲームは数多く作られています。例えば『ヘクシオン』は、『テトリス』のブロックを六角形状にしたものです。また「3次元のブロックを落とす」(→p. 123) で紹介する『ブロックアウト』は、『テトリス』を立体化したようなゲームです。完全な3次元ではなく、2次元のステージを立体的に並べた『ウェルトリス』というゲームもあります。

アルゴリズム

ブロックを落とすアクションは、ブロックやステージをセルで表現すると、比較的簡単に実現できます。セルについては「迷路を歩く」(→p. 10) を参照してください。

まず、落ちてくるブロックをセルで表現します (Fig. 2-6)。ここでは「4×4」のセルを使うことにしましょう。矩形ブロックのセルは文字「#」で表し、空のセルは空白文字で表します。

Fig. 2-6　ブロックをセルで表現する

ブロックを
セルで表現する

	#	#	
	#	#	

矩形ブロックのセル
(#)

空のセル
(空白)

他のブロックも同様に
4×4のセルで表現する

#	#	#	#

#	#		
	#	#	

		#	
#	#	#	

	#		
#	#	#	

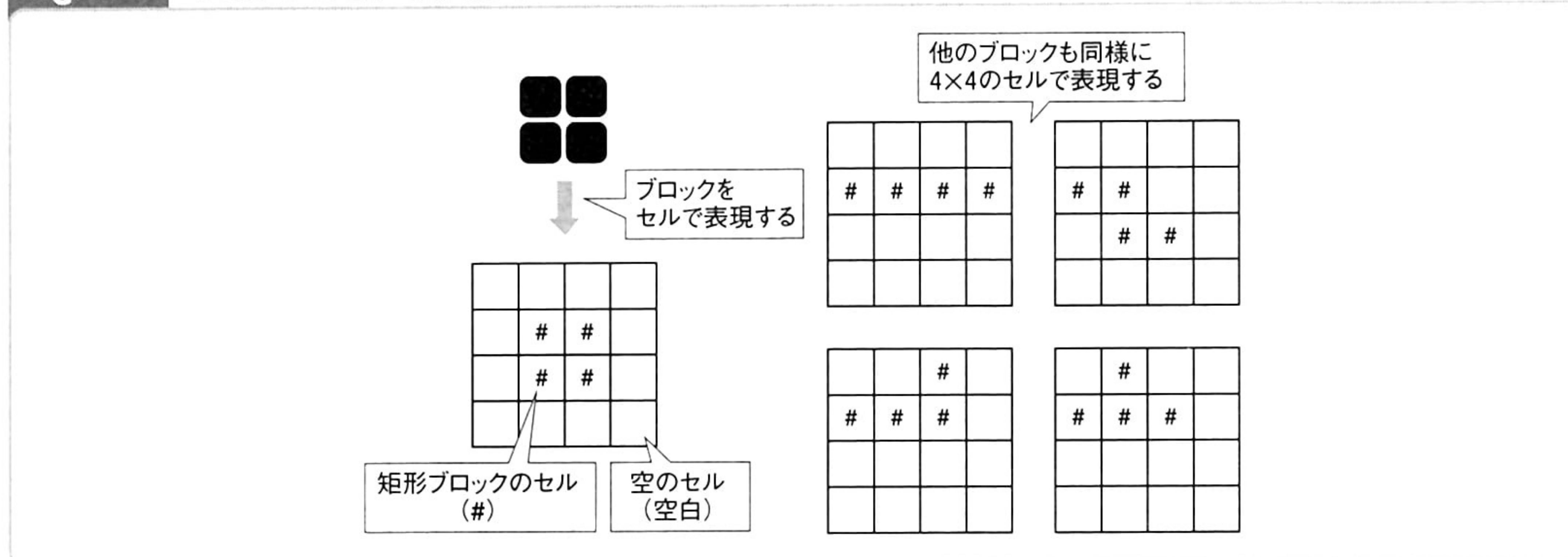
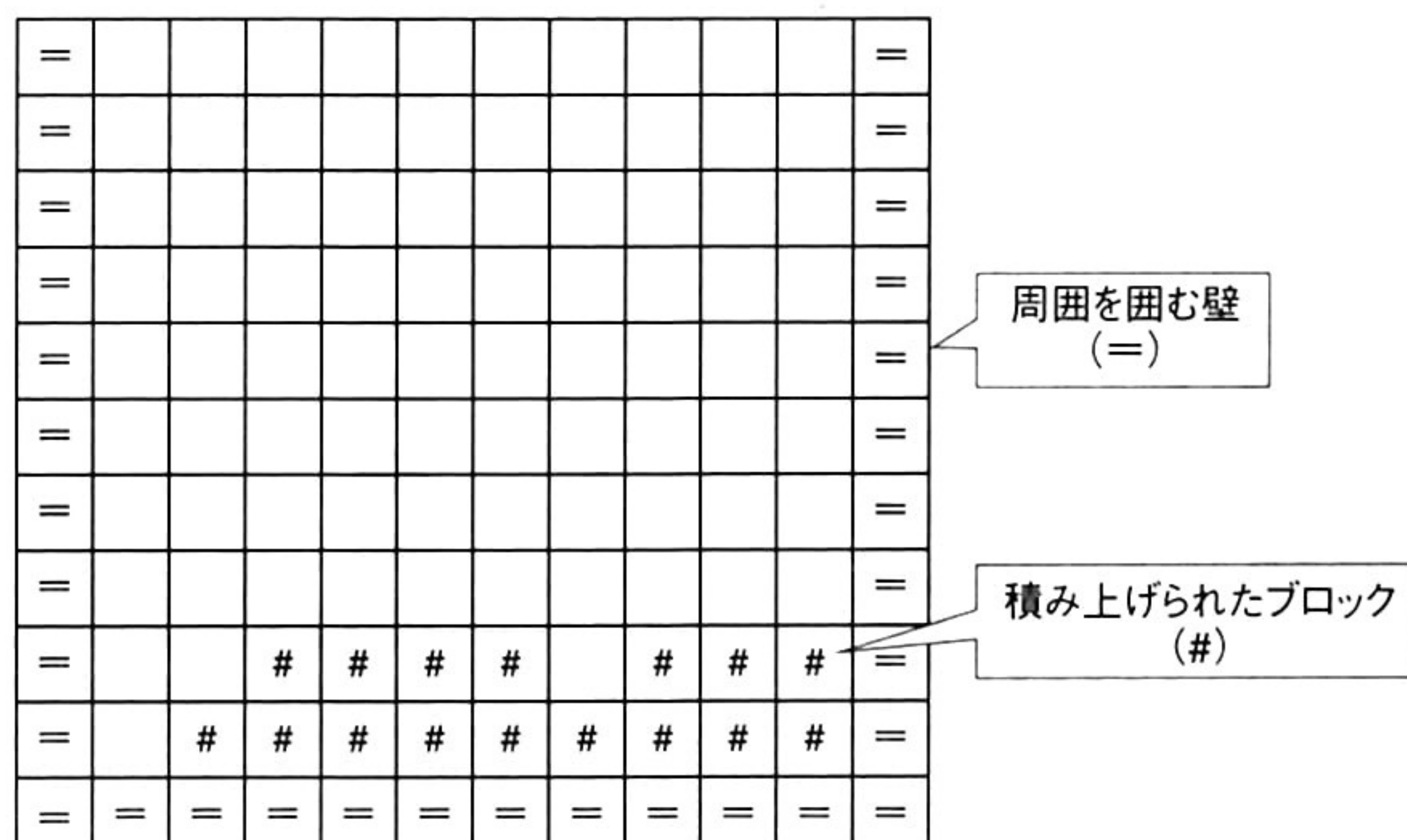



Fig. 2-7 ステージをセルで表現する

=											=
=											=
=											=
=											=
=											=
=											=
=											=
=											=
=			#	#	#	#		#	#	#	=
=		#	#	#	#	#	#	#	#	#	=
=	=	=	=	=	=	=	=	=	=	=	=

周囲を囲む壁（=）
積み上げられたブロック（#）

次に、ステージもセルで表現します（Fig. 2-7）。ここではステージの周囲を囲む壁を、文字「=」で表すことにします。積み上げられたブロックは、文字「#」で表しました。

落下速度を変化させる

ブロックを時間とともに落下させるには、タイマーを使います。定期的にタイマーを更新しておき、タイマーが一定値になるたびに、ブロックのセル座標（セル単位の座標）を更新しま

Fig. 2-8 時間とともにブロックを落下させる

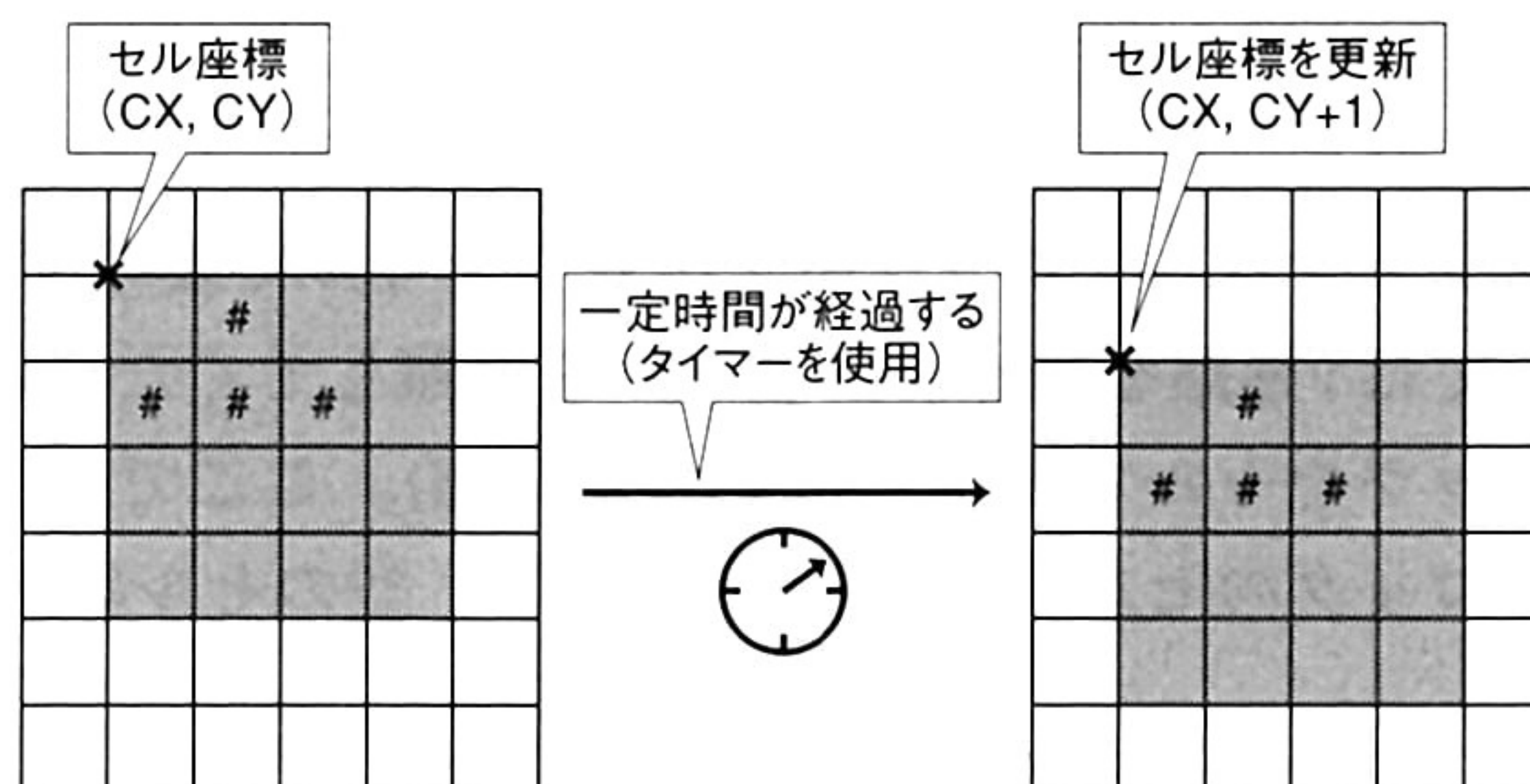
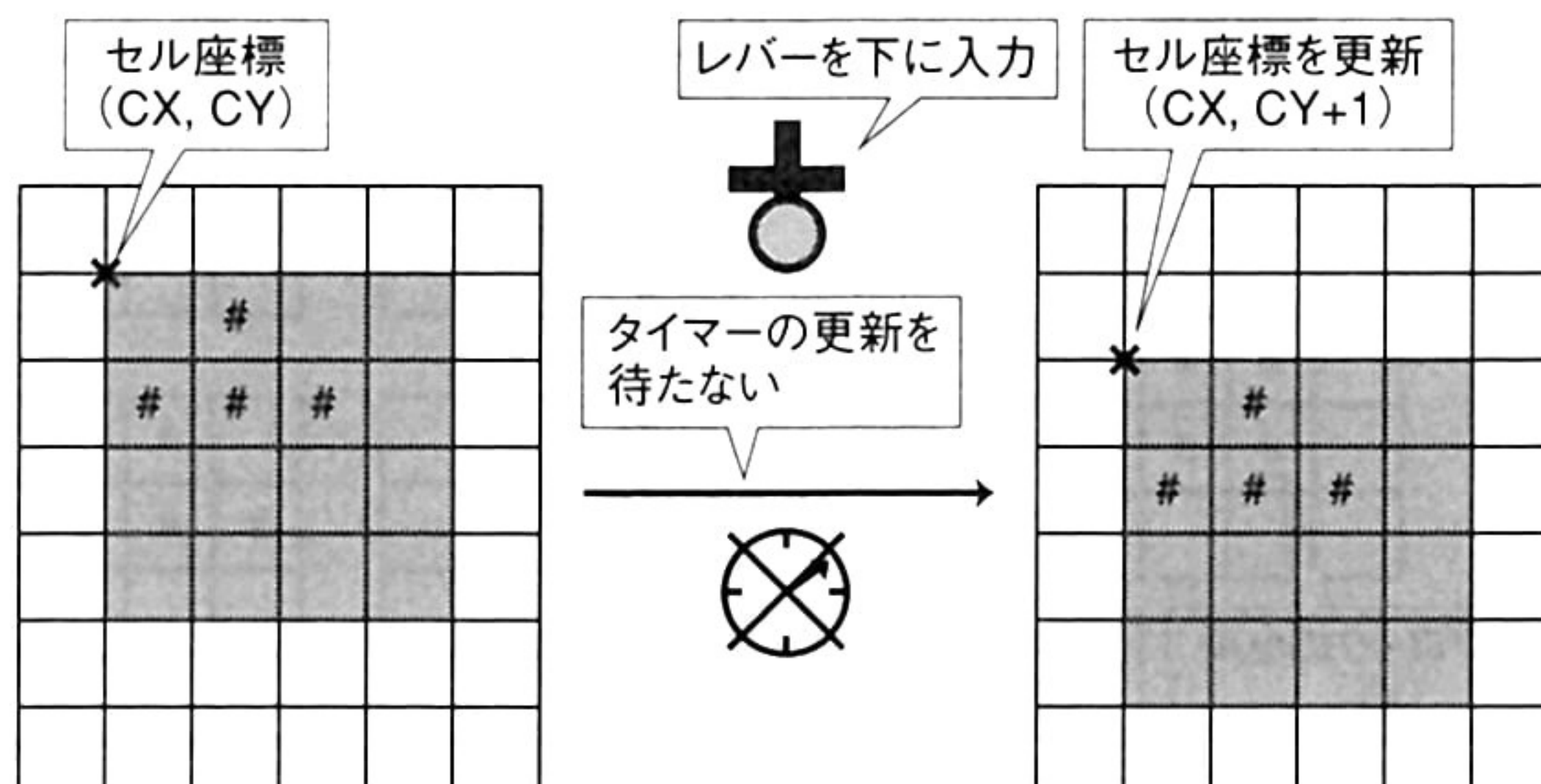



Fig. 2-9 レバー入力でブロックの落下スピードを上げる

す。例えばセル座標が(CX, CY)の場合には、CYに「1」を加算して、座標を(CX, CY+1)にします(Fig. 2-8)。

レバーを入力したときにブロックの落下スピードを上げるには、タイマーの値にかかわらず、即座にセル座標を更新します(Fig. 2-9)。通常時はタイマーの更新を待って落下させますが、レバー入力時はタイマーを待たないため、ブロックの落下スピードが上がります。

ブロックの接触判定

セル座標を更新する際には、ブロックのセルとステージのセルとの間で、当たり判定処理を行います。これは、セルを使った当たり判定処理です。なお、「統合して形を作る」(→p. 166)や「ボールを軌道に撃ち込む」(p. 294)では、座標を作った当たり判定処理を使っています。

同じ位置にあるセルの内容を調べて、両方のセルが空でない場合(両方のセルに何かがある場合)は、接触したと判定します。ブロックがステージの壁や他のブロックに接触していなければ、落下させることができます(Fig. 2-10)。

Fig. 2-10 ブロックとステージのセルを使った当たり判定処理

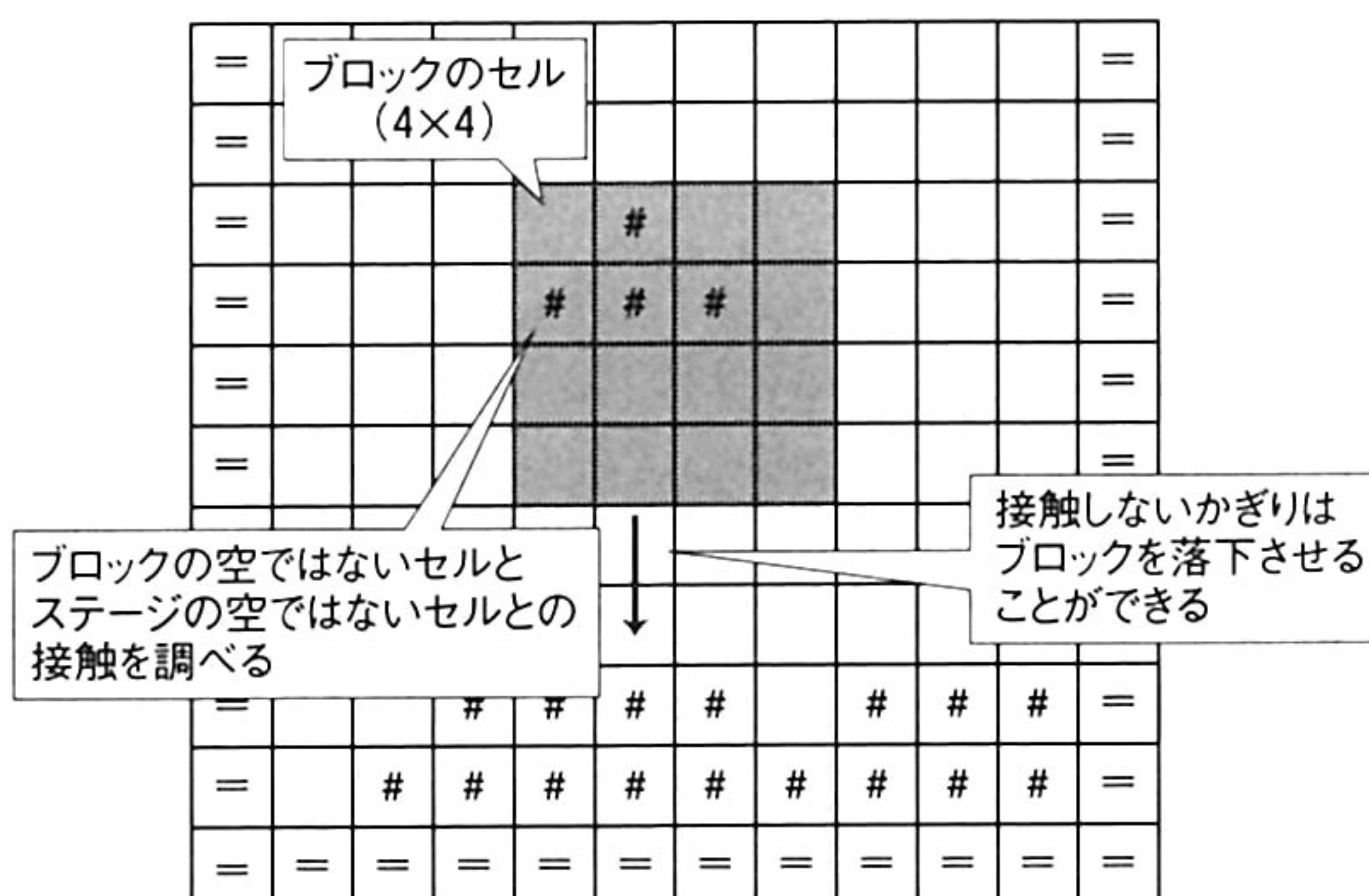



Fig. 2-11 ブロックを着地させる

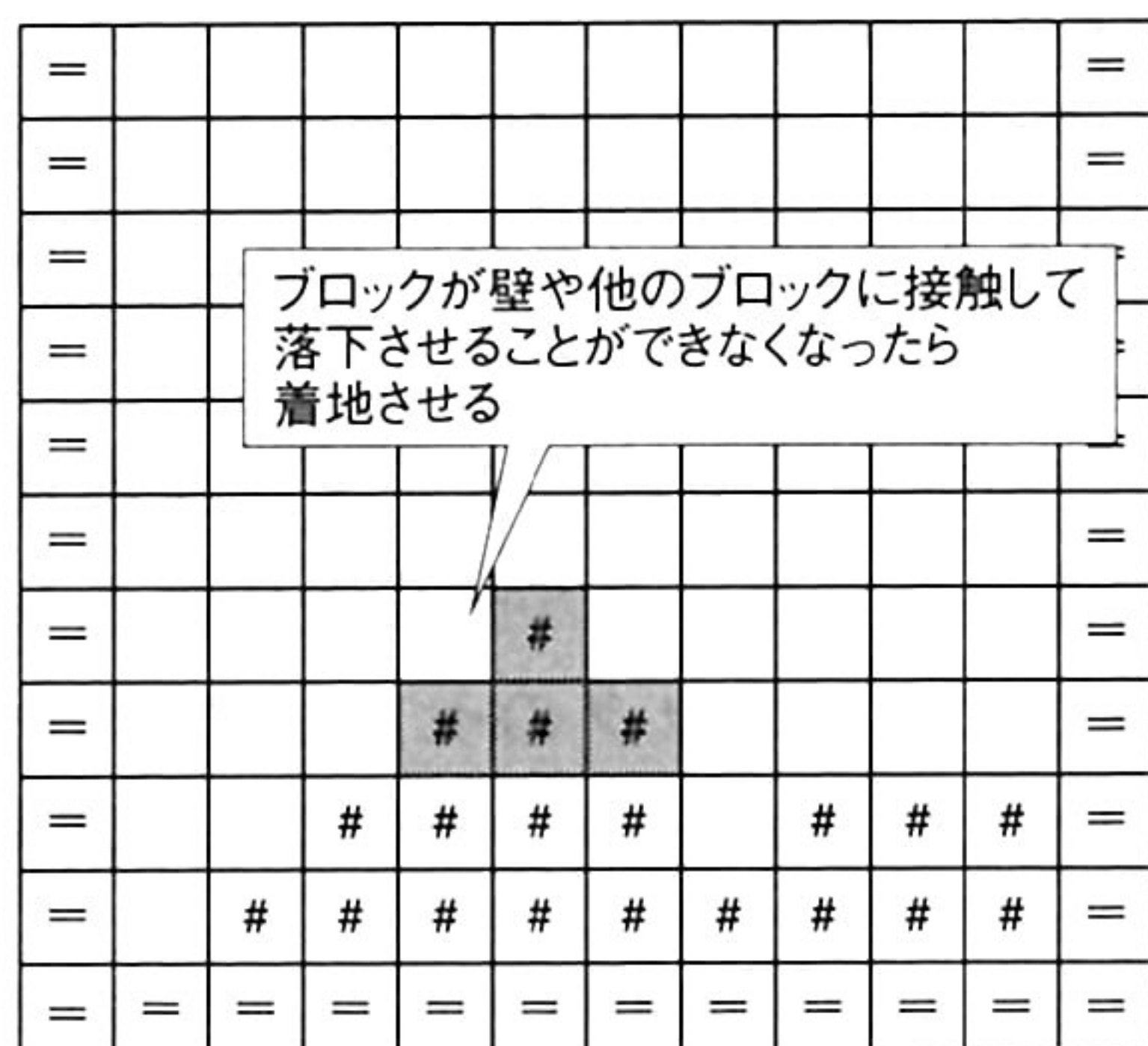
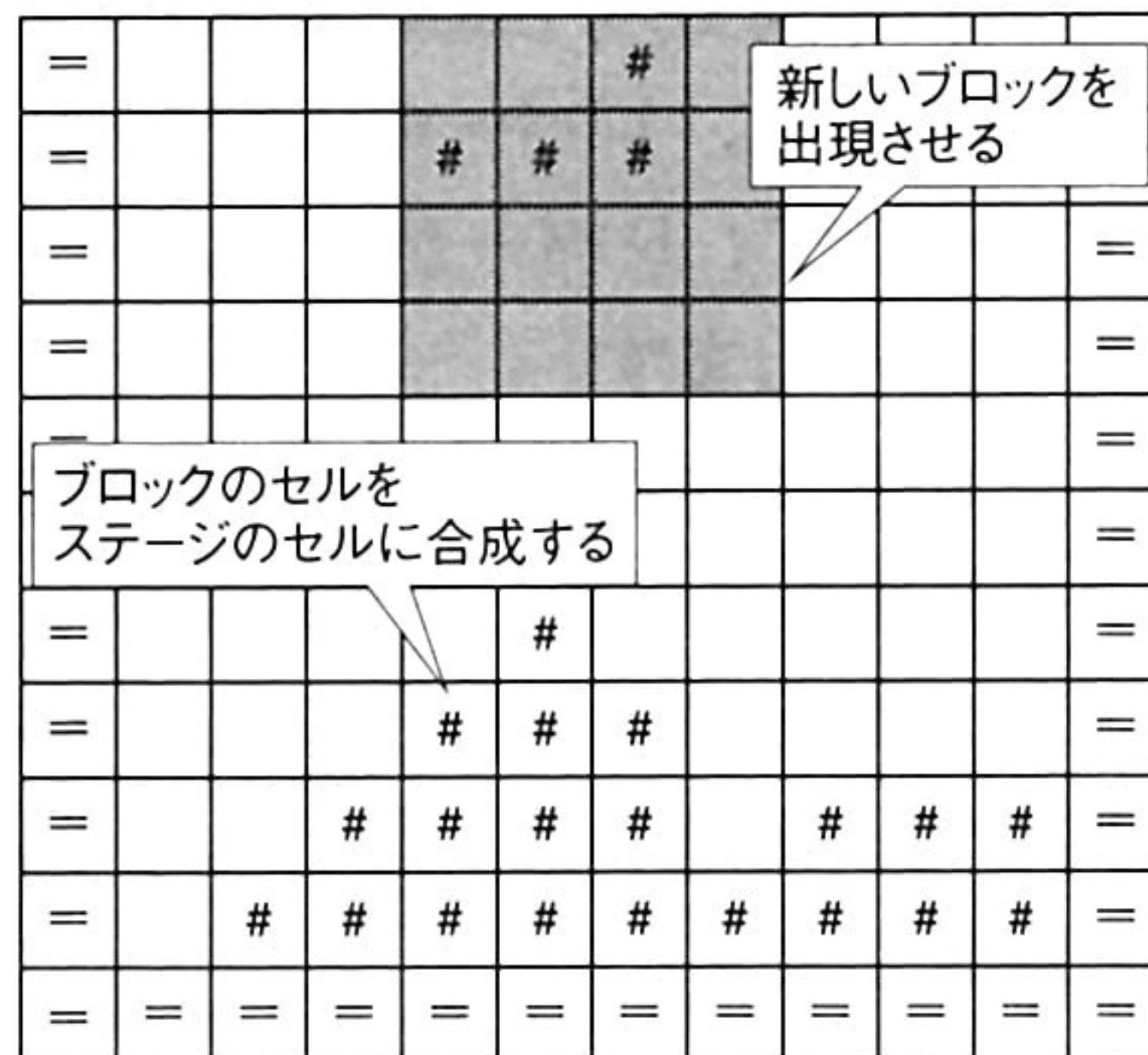



Fig. 2-12 ステージのセルにブロックのセルを合成する

もしもブロックがステージの壁や他のブロックに接触していたら、落下させることができないので、その場所に着地させます(Fig. 2-11)。

ブロックが着地したら、ステージのセルにブロックのセルを合成します(Fig. 2-12)。これでブロックがステージ上に残り、次々にブロックを積み上げていくことができます。そして、画面上方には新しいブロックを出現させます。

セル座標の更新

ブロックのセル座標を更新する際には、画面上でもブロックを落下させるために、描画座標(ブロックを描画する座標)も更新します。ここでは描画座標を(X, Y)としましょう。セル座標は(CX, CY)から(CX, CY+1)に変化しますが、ブロックを滑らかに動かすためには、描画座標は少しずつ変化させるとよいでしょう。例えば、

(X, Y+0.1)
(X, Y+0.2)
(X, Y+0.3)
⋮

といった具合に、ある程度の時間を使って、描画座標を目標値に近づけます。

プログラム

List 2-1はブロックを落とすプログラムです。セルの当たり判定処理、セルの合成処理、およびブロックの移動処理を掲載しました。

セルの当たり判定処理(Hit関数)は、ブロックのセルとステージのセルの当たり判定処理に使います。ブロックがステージの壁や他のブロックに接触したかどうかを判定します。

セルの合成処理(Merge関数)は、ブロックのセルをステージのセルに合成するために使います。セルを合成することによって、ブロックをステージ上に残すことができます。

ブロックの移動処理(Move関数)では、ブロックの状態を入力状態と移動状態に分けています。入力状態では、レバーの入力を受け付けたり、落下タイマーを更新したりといった処理を行います。そして、レバーが下に入力されたときや、落下タイマーが一定値に達したときには、ブロックを落とします。

ブロックが落とせる場合には、移動状態に移行します。移動状態では、タイマーを利用して、ブロックの描画座標を少しずつ変化させます。これで、画面上をブロックが滑らかに落下します。

ブロックが落とせない場合には、ブロックを着地させます。ブロックのセルをステージのセルに合成して、着地状態に移行します。着地状態については「ブロックを1段揃えて消す」(→p. 64)で解説します。

List 2-1 ブロックを落とす（CCellクラス、CDroppingBlockBlockクラス）

```
// セルの当たり判定処理
// セルの集まり（cell）を座標（x, y）に配置したときに、
// 空ではないセル同士が重なるかどうかを調べる
// 重なる場合にはtrue、重ならない場合にはfalseを返す
bool CCell::Hit(int x, int y, CCell* cell) {
    for (int i=0, in=cell->GetYSize(); i<in; i++) {
        for (int j=0, jn=cell->GetXSize(); j<jn; j++) {

            // 空白ではないセル同士が1つでも重なったら
            // trueを返す
            if (cell->Get(j, i)!=' ' && Get(x+j, y+i)!=' ') {
                return true;
            }
        }
    }

    // 空白ではないセル同士が1つも重ならなかったら
    // falseを返す
    return false;
}

// セルの合成処理
// セルの集まり（cell）を座標（x, y）に合成する
// 空のセルは合成しない
void CCell::Merge(int x, int y, CCell* cell) {
    for (int i=0, in=cell->GetYSize(); i<in; i++) {
        for (int j=0, jn=cell->GetXSize(); j<jn; j++) {

            // 空ではないセルだけを合成する
            if (cell->Get(j, i)!=' ') {
                Set(x+j, y+i, cell->Get(j, i));
            }
        }
    }
}

// ブロックの移動処理
bool CDroppingBlockBlock::Move(const CInputState* is) {

    // 入力状態の処理
    if (State==0) {

        // 落下タイマーの更新
        DropTime++;

        // レバーを下に入力するか、
        // 落下タイマーが一定値に達したら、ブロックを落下させる
        if ((is->Down && !PrevDown)|| DropTime==60) {
```

```
			// セル座標を更新したときに、
			// ブロックがステージの壁や他のブロックに
			// 接触するかどうかを調べる
			if (StageCell->Hit(CX, CY+1, CurrentBlock[Turn])) {

				// 接触する場合には、
				// ブロックのセルをステージのセルに合成する
				StageCell->Merge(CX, CY, CurrentBlock[Turn]);

				// 着地状態に移行する
				State=2;
			} else {

				// 接触しない場合には、
				// 落下タイマーや速度を設定し、
				// セル座標を更新する
				DropTime=0;
				CY++;
				VX=0;
				VY=1;

				// 移動状態に移行する
				Time=0;
				State=1;
			}
		} else
		// ...（中略）...
	}

	// 移動状態の処理
	if (State==1) {

		// タイマーの更新
		// ここではタイマーを増加させているが、
		// 減少させる処理にしてもよい（Stage01のプログラムを参照）
		Time++;

		// タイマーを使って、描画座標を少しずつ更新する
		X=CX-VX*(1-Time*0.1f);
		Y=CY-VY*(1-Time*0.1f);

		// タイマーが一定値に達したら、入力状態に移行する
		if (Time==10) {
			State=0;
		}
	}
}
```

SAMPLE

「DROPPING BLOCK」は「ブロックを落とす」「ブロックを左右に移動する」「ブロックを回転させる」「ブロックを1段揃えて消す」「次のブロックを表示する」「落下予測位置を表示する」のサンプルです。

レバーの左右（カーソルキーの左右）でブロックが移動し、レバーの下（カーソルキーの下）でブロックの落下スピードが上がります。ボタン0（Zキー）を押すとブロックが回転します。

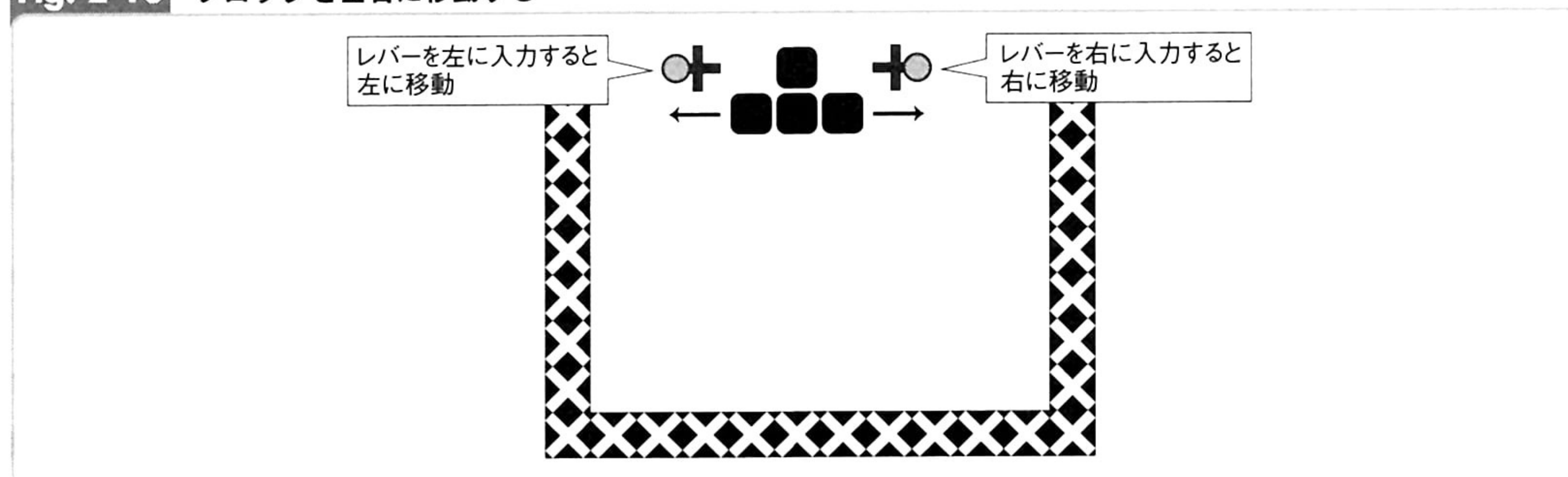
ブロックを横1段に隙間なく詰めると、ブロックを消すことができます。画面右端に表示されているブロックは、次に落ちてくるブロックです。また、灰色で表示されているのはブロックの落下予測位置です。

DROPPING BLOCK → p. 386

ブロックを左右に移動する

時間とともに落下するブロックを、左右に移動するアクションです。真下にブロックを落とすだけではなく、落下位置を選ぶことで、思った形にブロックを積み上げることができます。

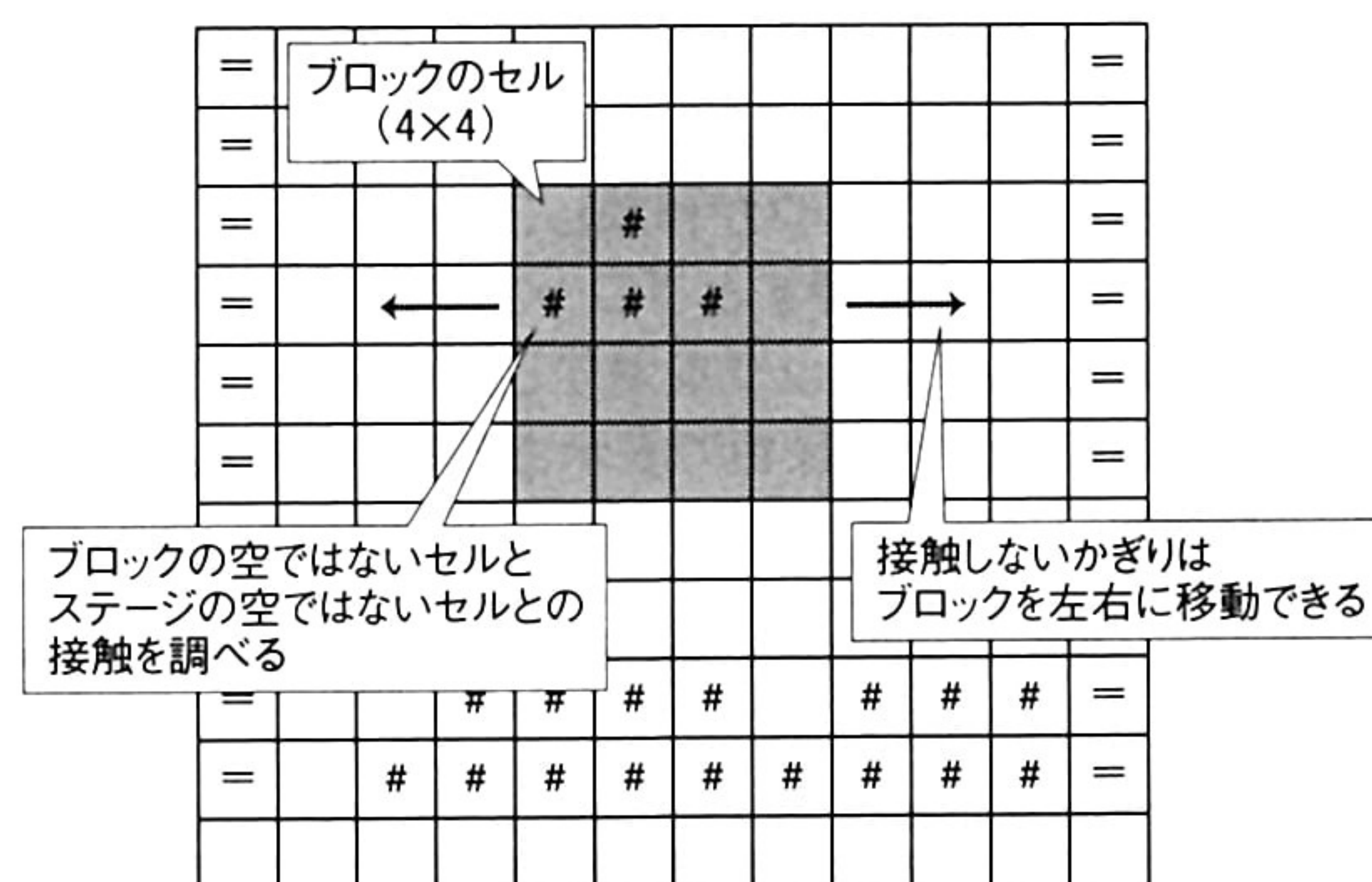
レバーを左に入力するとブロックは左へ、右に入力すると右へ移動します（Fig. 2-13）。なお、ステージの端には壁があるので、それ以上は移動できません。

Fig. 2-13 ブロックを左右に移動する

アルゴリズム

ブロックを左右に移動するには、レバー入力に応じて、ブロックのセル座標を更新します。このとき、ブロックのセルとステージのセルとの間で当たり判定処理を行い、ブロックがステージの壁や他のブロックに接触するかどうかを調べます（Fig. 2-14）。

接触する場合には、ブロックを移動することはできません。接触しない場合には、ブロックのセル座標を更新します。

Fig. 2-14 ブロックを左右に移動するための当たり判定処理

ブロックのセル
(4×4)
ブロックの空ではないセルと
ステージの空ではないセルとの
接触を調べる
接触しないかぎりは
ブロックを左右に移動できる

セル座標を更新したら、描画座標も更新します。描画座標を (X, Y) とすると、

(X+0.1, Y)
(X+0.2, Y)
(X+0.3, Y)
⋮

のように、少しずつ更新します。これで「ブロックを落とす」(→p. 50) と同様に、ブロックが滑らかに左右へ移動します。

プログラム

List 2-2はブロックを左右に移動するプログラムです。ブロックの移動処理から、関連する部分を抜粋しました。

ブロックには入力状態と移動状態があります。入力状態ではレバーの入力を受け付け、ブロックのセルとステージのセルとの間で当たり判定処理を行います。ブロックがステージに接触していなければ、セル座標を更新し、移動状態に移行します。

移動状態では、タイマーを使って、ブロックの描画座標を少しずつ変化させます。これはブロックを落とすアクションと共通の処理です。

List 2-2 ブロックを左右に移動する (CDroppingBlockBlockクラス)

```
// ブロックの移動処理
bool CDroppingBlockBlock::Move(const CInputState* is) {

    // 入力状態の処理
    if (State==0) {
```

```
		// ...（中略）...

		// レバーを左に入力しており、
		// かつブロックが移動先でステージに接触しなければ、
		// ブロックを左に移動させる
		if (
			is->Left &&
			!StageCell->Hit(CX-1, CY, CurrentBlock[Turn])
		) {
			// セル座標の更新
			CX--;

			// 速度の設定
			VX=-1;
			VY=0;

			// 移動状態に移行する
			Time=0;
			State=1;
		} else

		// レバーを右に入力しており、
		// かつブロックが移動先でステージに接触しなければ、
		// ブロックを右に移動させる
		if (
			is->Right &&
			!StageCell->Hit(CX+1, CY, CurrentBlock[Turn])
		) {
			// セル座標の更新
			CX++;

			// 速度の設定
			VX=1;
			VY=0;

			// 移動状態に移行する
			Time=0;
			State=1;
		} else
		// ...（中略）...
	}

	// 移動状態の処理
	if (State==1) {

		// タイマーの更新
		Time++;

		// タイマーを使って、描画座標を少しずつ更新する
```

```
        X=CX-VX*(1-Time*0.1f);
        Y=CY-VY*(1-Time*0.1f);

        // タイマーが一定値に達したら、入力状態に移行する
        if (Time==10) {
            State=0;
        }
    }
    // …（中略）…
}
```

ブロックを回転させる

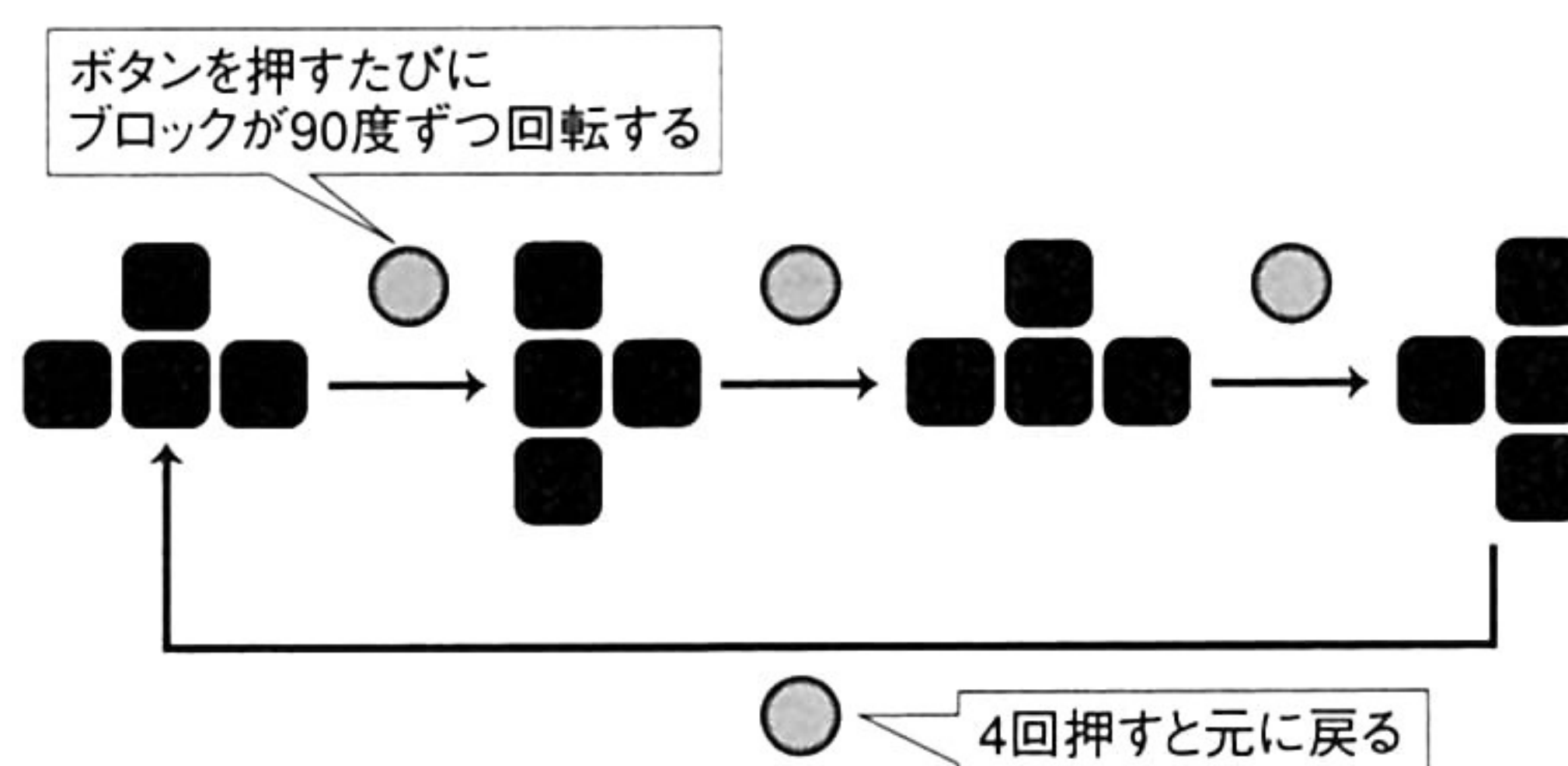
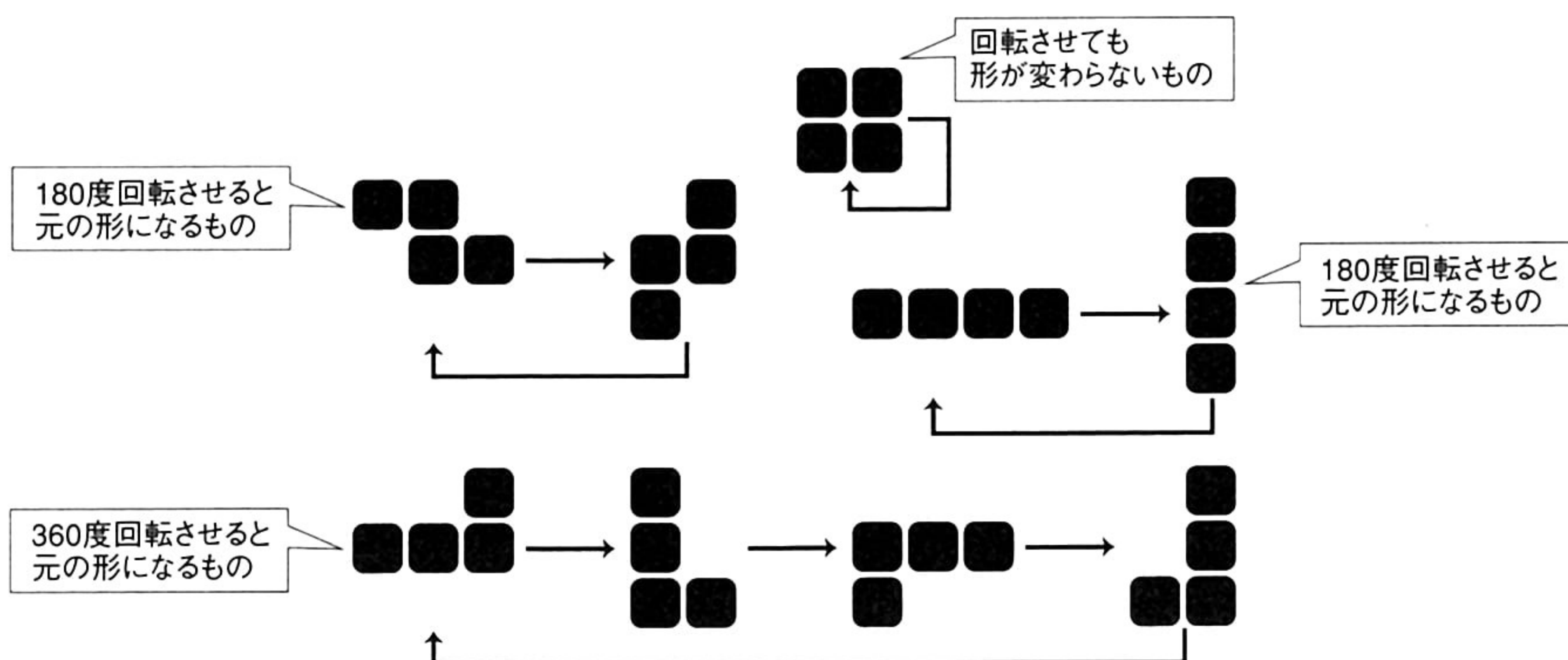
ボタン入力でブロックを回転させるアクションです。左右への移動に加えて、ブロックを回転させることによって、より隙間なくブロックを積み上げることができます。

ボタンを1回押すたびに、ブロックは90度ずつ回転します（Fig. 2-15）。4回ボタンを押すと、

Fig. 2-15 ブロックの回転

ボタンを押すたびに
ブロックが90度ずつ回転する
4回押すと元に戻る

Fig. 2-16 いろいろなブロックの回転

回転させても
形が変わらないもの
180度回転させると
元の形になるもの
180度回転させると
元の形になるもの
360度回転させると
元の形になるもの

ブロックは元の角度に戻ります。ここではブロックが右に回転することにしましたが、ゲームによっては左に回転する場合や、ボタンに応じて左右の回転方向を選べる場合もあります。

ブロックにはいろいろな形があります。どのブロックも同様に、ボタンを1回押すたびに90度ずつ回転します(Fig. 2-16)。ただし、ブロックの形によっては、回転させても形が変わらないものや、180度回転させると元の形に戻るものもあります。

アルゴリズム

ブロックの回転を実現するには、90度ごとにブロックのパターンを用意しておきます(Fig. 2-17)。ここではブロックを「4×4」のセルで表現しました。

90度単位の回転で、360度ぶんのパターンが必要なので、ブロック1種類ごとに4つのパターンが必要になります。回転させても形が変わらないものや、180度回転させると元の形に戻るものに関しても、4つのパターンを用意しておくと処理が簡単です(Fig. 2-18)。

Fig. 2-17 ブロックのパターン

90度ごとに
ブロックのパターンを
用意しておく

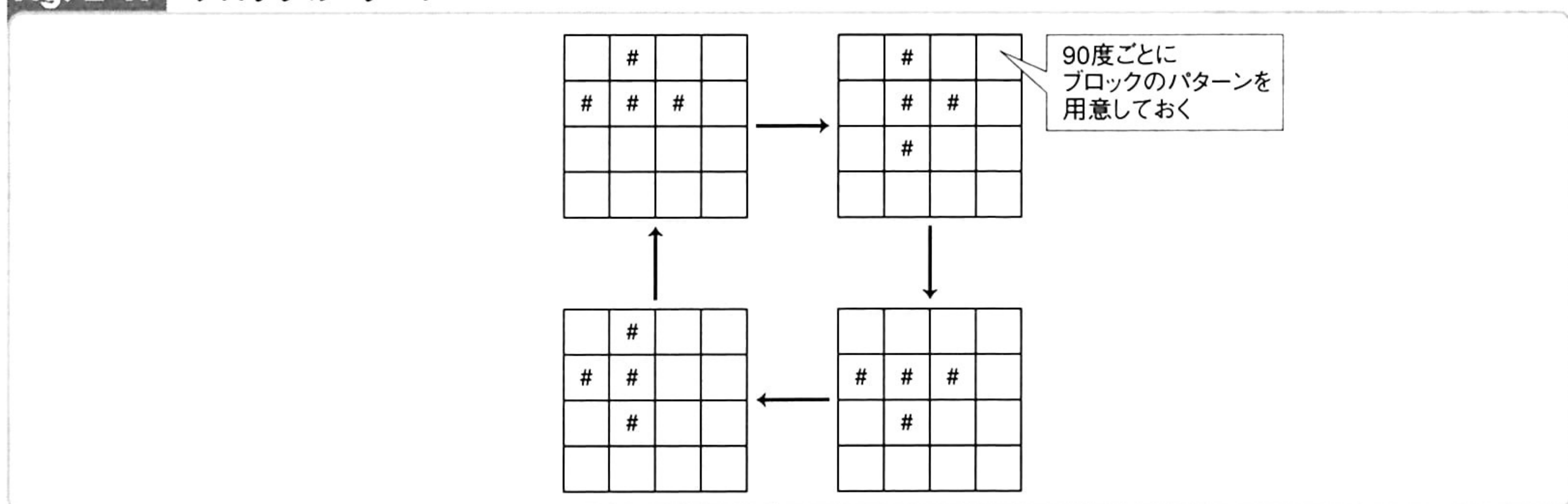
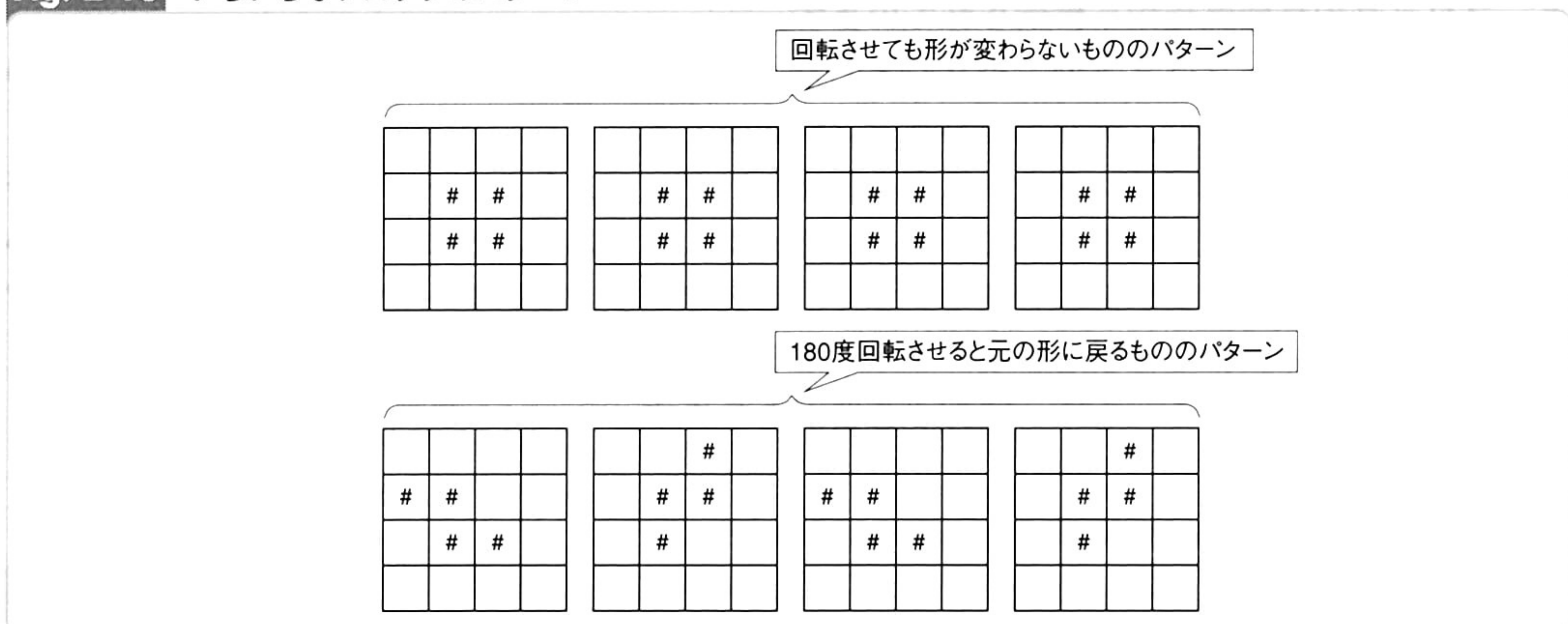
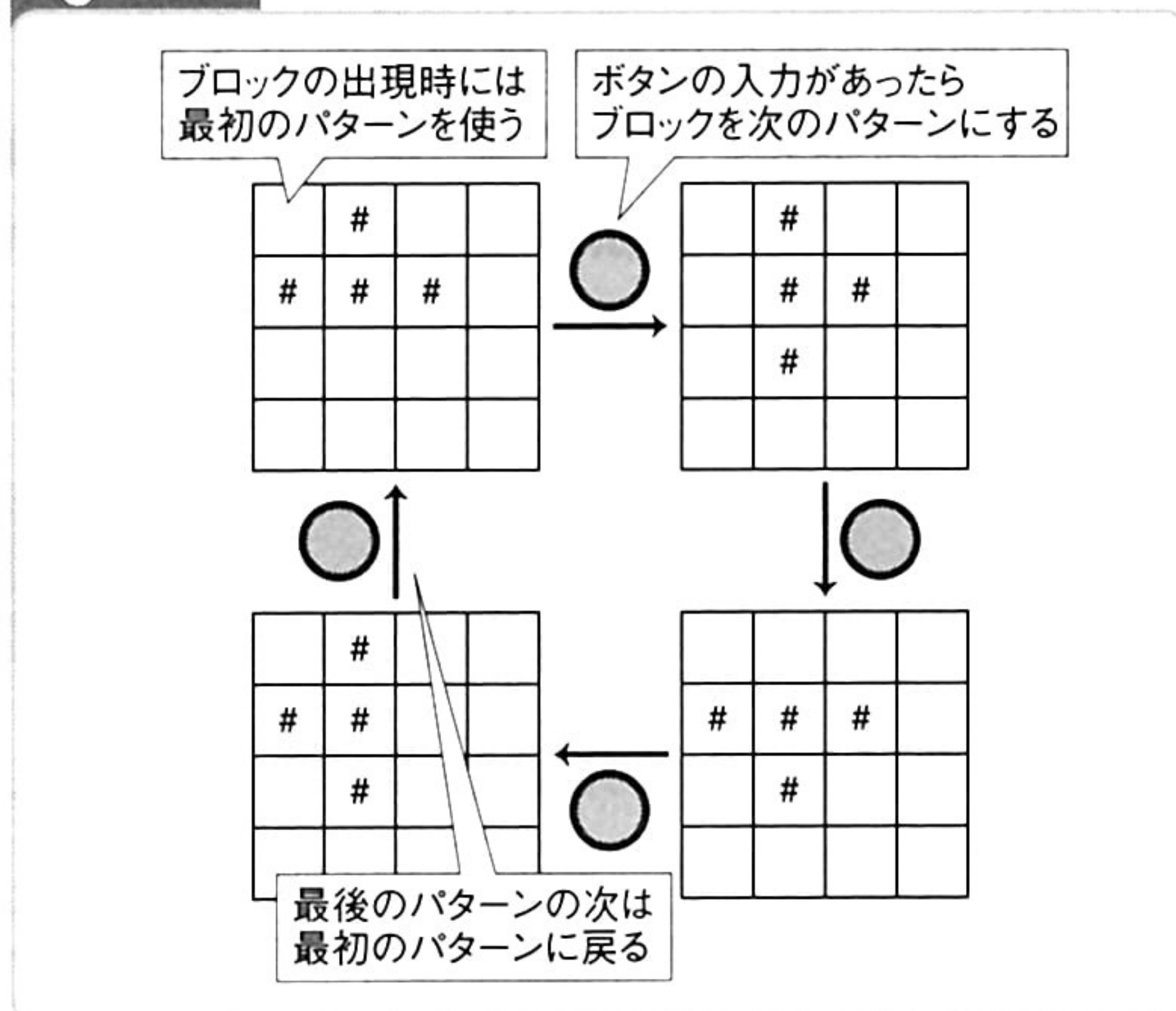
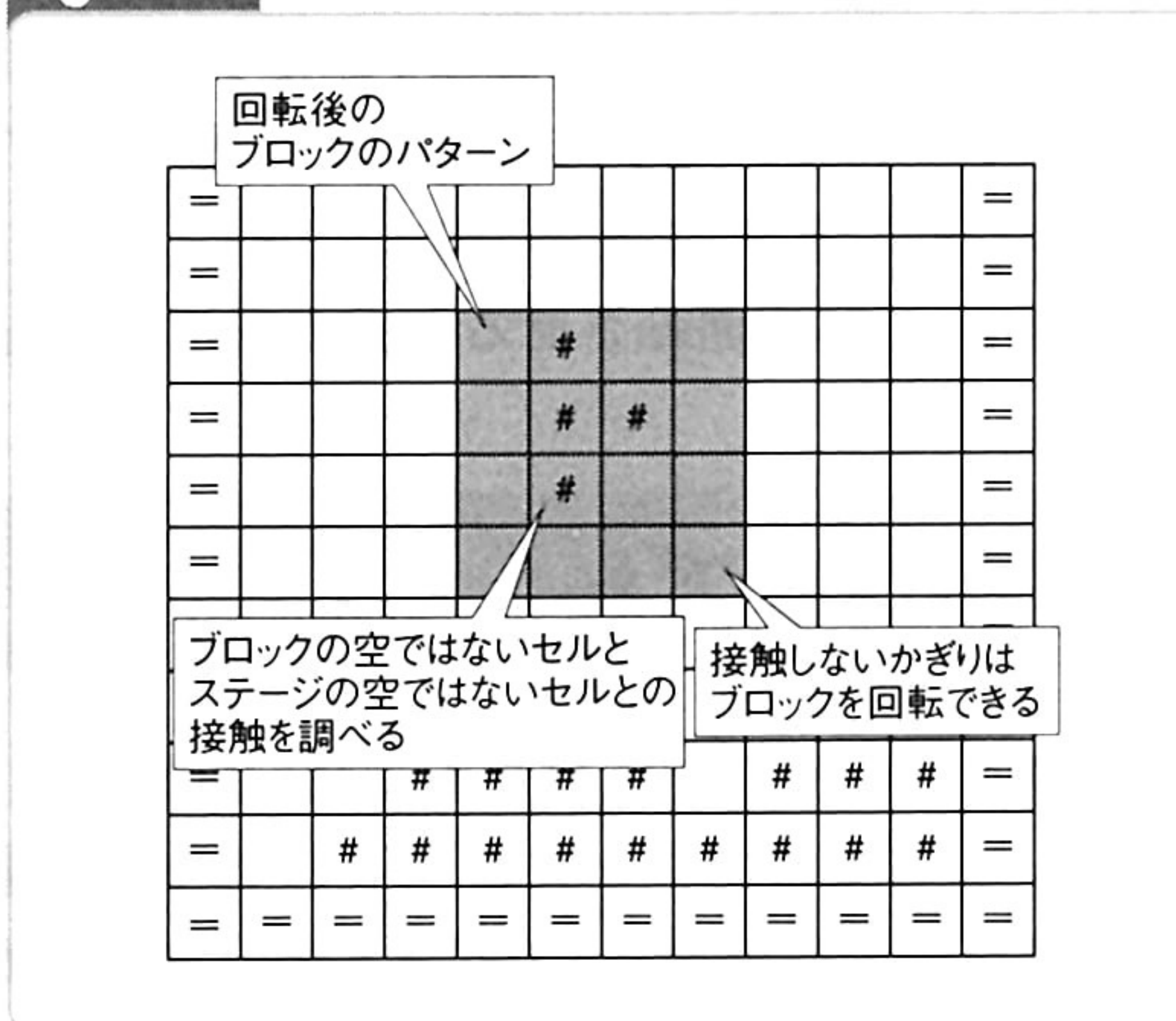
Fig. 2-18 いろいろなブロックのパターン

回転させても形が変わらないもののパターン

180度回転させると元の形に戻るもののパターン

Fig. 2-19 パターンの変化

ブロックの出現時には
最初のパターンを使う
ボタンの入力があったら
ブロックを次のパターンにする
最後のパターンの次は
最初のパターンに戻る

Fig. 2-20 回転できるかどうかの判定処理

回転後の
ブロックのパターン
ブロックの空ではないセルと
ステージの空ではないセルとの
接触を調べる
接触しないかぎりは
ブロックを回転できる

ブロックの出現時には、4つのパターンのうち最初のパターンを使います。ボタンの入力があったら、ブロックを次のパターンに変化させます（Fig. 2-19）。最後のパターンの次は、最初のパターンに戻ります。

ブロックを回転させるときには、回転ができるかどうかを判定する必要があります。そのためには、回転後のブロックのセルと、ステージのセルとの間で当たり判定処理を行います（Fig. 2-20）。もしも接触する場合には、回転後のブロックがステージの壁や他のブロックにぶつかるということなので、ブロックを回転させることはできません。接触しなければ、パターンを変化させて、ブロックを回転させます。

プログラム

List 2-3はブロックを回転させるプログラムです。ブロックのパターンをデータ化したものと、ブロックの移動処理から構成されています。

ブロックのパターンは文字列で用意しました。ブロックが7種類あり、ブロックごとに4つの回転パターンがあるので、計28個のパターンが必要です。

ブロックの移動処理では、ボタン入力を検出し、ブロックとステージの当たり判定処理を行って、ブロックを回転させます。ここでは、ボタン入力があり、かつ直前のボタン入力がなかった場合だけ、ブロックを回転させています。これはボタンを押しっぱなしにしたときに、ブロックが連続的に回転してしまわないようにするための処理です。ブロックを回転させるには、ボタンを一度離してから、もう一度押す必要があります。

List 2-3 ブロックを回転させる（CDroppingBlockBlockクラス）

```
// ブロックのパターン
// ブロックの形状ごとに4つのパターンを用意する
```

```
char* DroppingBlockPattern[
    DROPPING_BLOCK_PATTERN_COUNT*
    DROPPING_BLOCK_TURN_COUNT
]={
    "    "
    " ## "
    " ## "
    "    "
    ,
    "    "
    " ## "
    " ## "
    "    "
    ,
    "    "
    " ## "
    " ## "
    "    "
    ,
    "    "
    " ## "
    " ## "
    "    "
    ,
    // ...（中略）...

    " #  "
    "### "
    "    "
    "    "
    ,
    " #  "
    " ## "
    " #  "
    "    "
    ,
    "    "
    "### "
    " #  "
    "    "
    ,
    " #  "
    "##  "
    " #  "
    "    "
};

// ブロックの移動処理
```

```
bool CDroppingBlockBlock::Move(const CInputState* is) {

    // 入力状態の処理
    if (State==0) {

        // ...（中略）...

        // ボタンを入力していて、直前にボタンを離しており、
        // かつ回転後にブロックとステージが接触しなければ、ブロックを回転させる
        if (
            is->Button[0] &&
            !PrevButton &&
            !StageCell->Hit(CX, CY,
                CurrentBlock[(Turn+1)%
                DROPPING_BLOCK_TURN_COUNT])
        ) {
            // ブロックを次のパターンに変化させる
            // 最後のパターンの次は、最初のパターンに戻す
            Turn=(Turn+1)%DROPPING_BLOCK_TURN_COUNT;
        }

        // 直前にボタンを押したかどうかを記録しておく
        PrevButton=is->Button[0];

        // レバーを下に入れ続けたときに、ブロックが次々に落ちないようにするための処理
        if (!is->Down) PrevDown=false;
    }
    // ...（中略）...
}
```

ブロックを1段揃えて消す

ブロックを横方向に1段揃えると、揃えた段のブロックが消えるというアクションです。移動や回転を利用して、ブロックを隙間なく積み上げ、次々に消していくことがゲームの目的です。

ブロックを消すには、ステージの左端から右端まで、ブロックを隙間なく詰めます(Fig. 2-21)。ブロックが1段揃うと、その段が消えます(Fig. 2-22)。図ではブロックがしだいに薄くなって消えていくように示しましたが、ブロックが飛び散るように消えるゲームや、特にエフェクトがなく消えるゲームもあります。

ブロックが消えると、消えた段よりも上に積まれていたブロックが落ちてきます(Fig. 2-23)。ブロックがステージの上端まで積もるとゲームオーバーなので、次々にブロックを揃えて消し、ブロックの山を低くしていく必要があります。

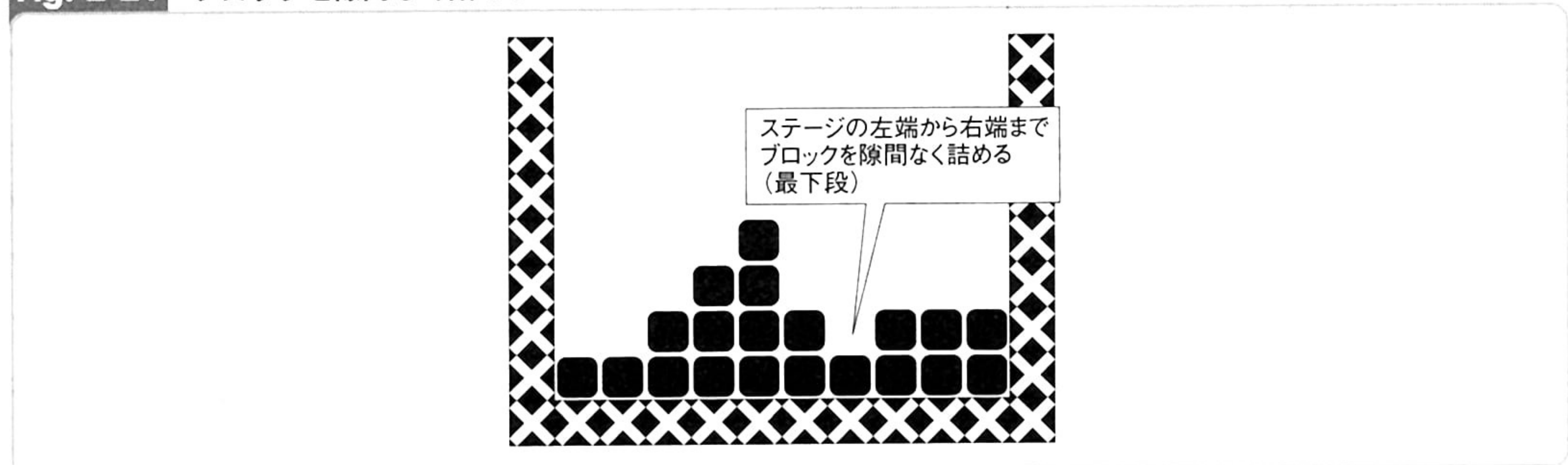
上手にブロックを積むと、1段ずつとはかぎらず、一度に複数の段をまとめて消すこともできます (Fig. 2-24)。消える段の間に消えない段があってもかまいません (Fig. 2-25)。

複数段をまとめて消すには、ブロックの山に細い隙間を作っておき、その隙間に細長いブロックを詰めるのがコツです。『テトリス』の場合には、最大で4段をまとめて消すことができます。

Fig. 2-21 ブロックを隙間なく詰める

ステージの左端から右端まで
ブロックを隙間なく詰める
（最下段）

Fig. 2-22 ブロックが揃った段が消える

隙間なく
ブロックが揃うと
その段は消える

ブロックの表示がしだいに
薄くなって消えていく
（ゲームによって演出が異なる）

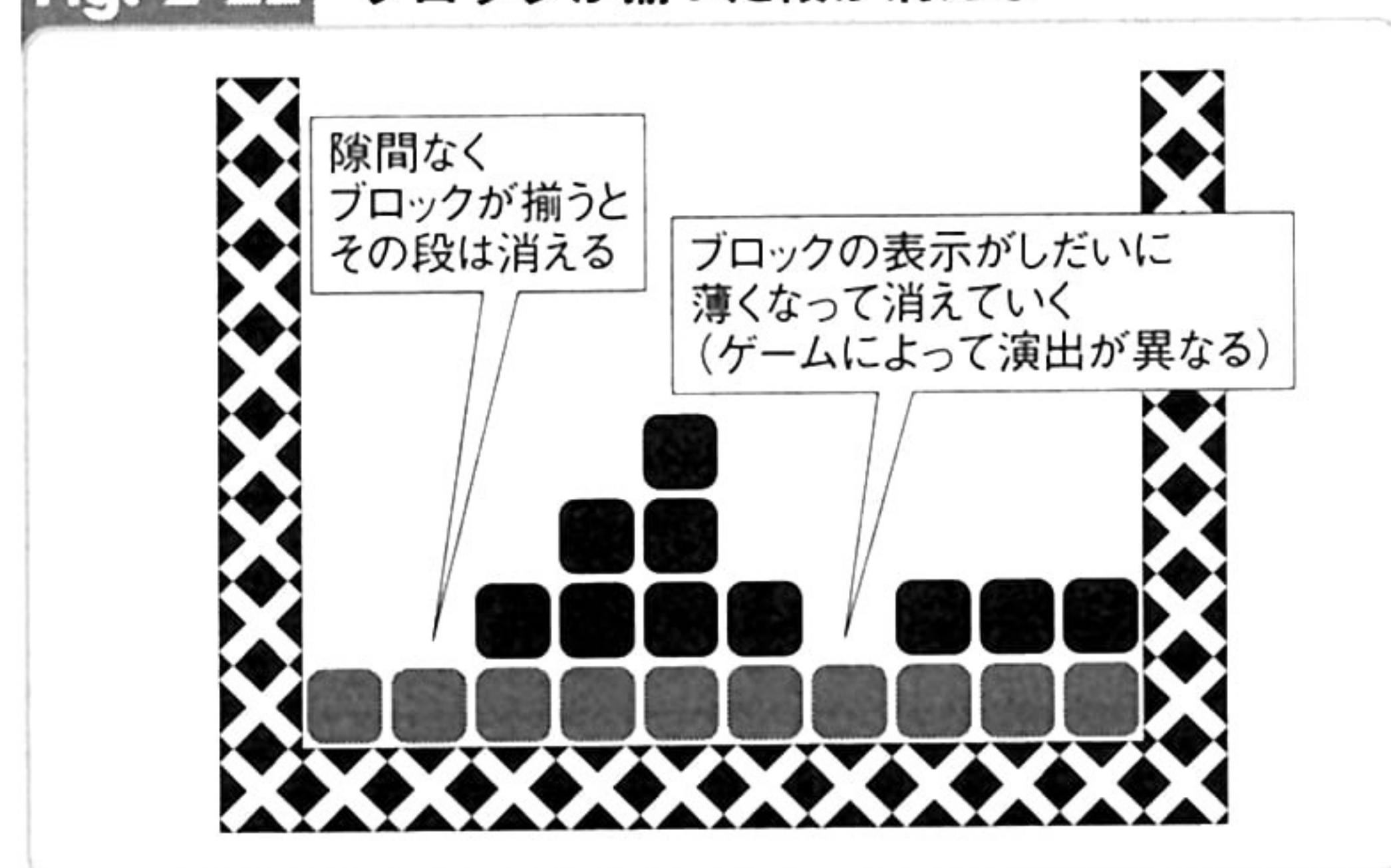



Fig. 2-23 上に積まれていたブロックが落ちてくる

上に積まれていたブロックが
落ちてくる

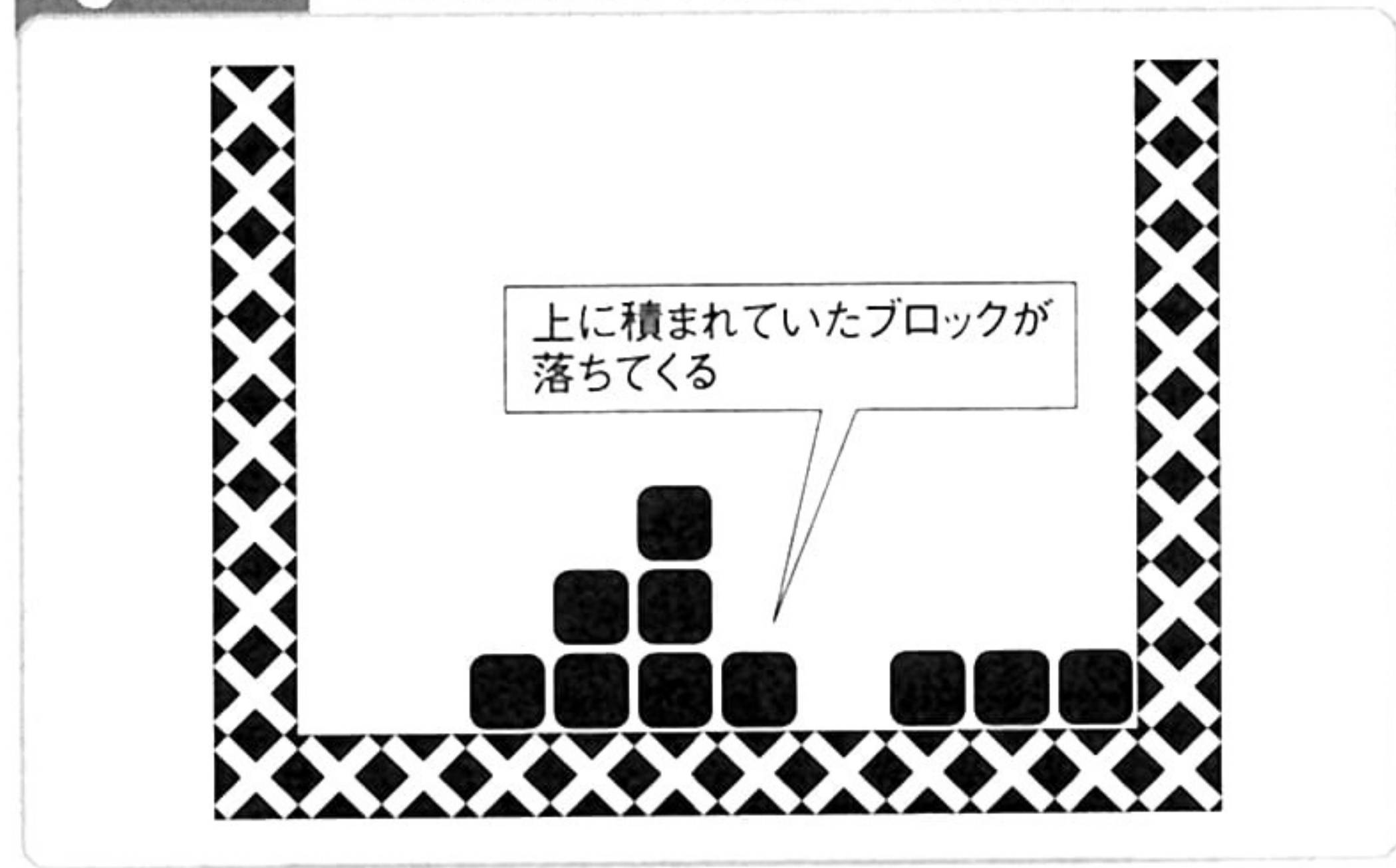



Fig. 2-24 複数の段をまとめて消す

複数の段
（4段）を
まとめて
消す

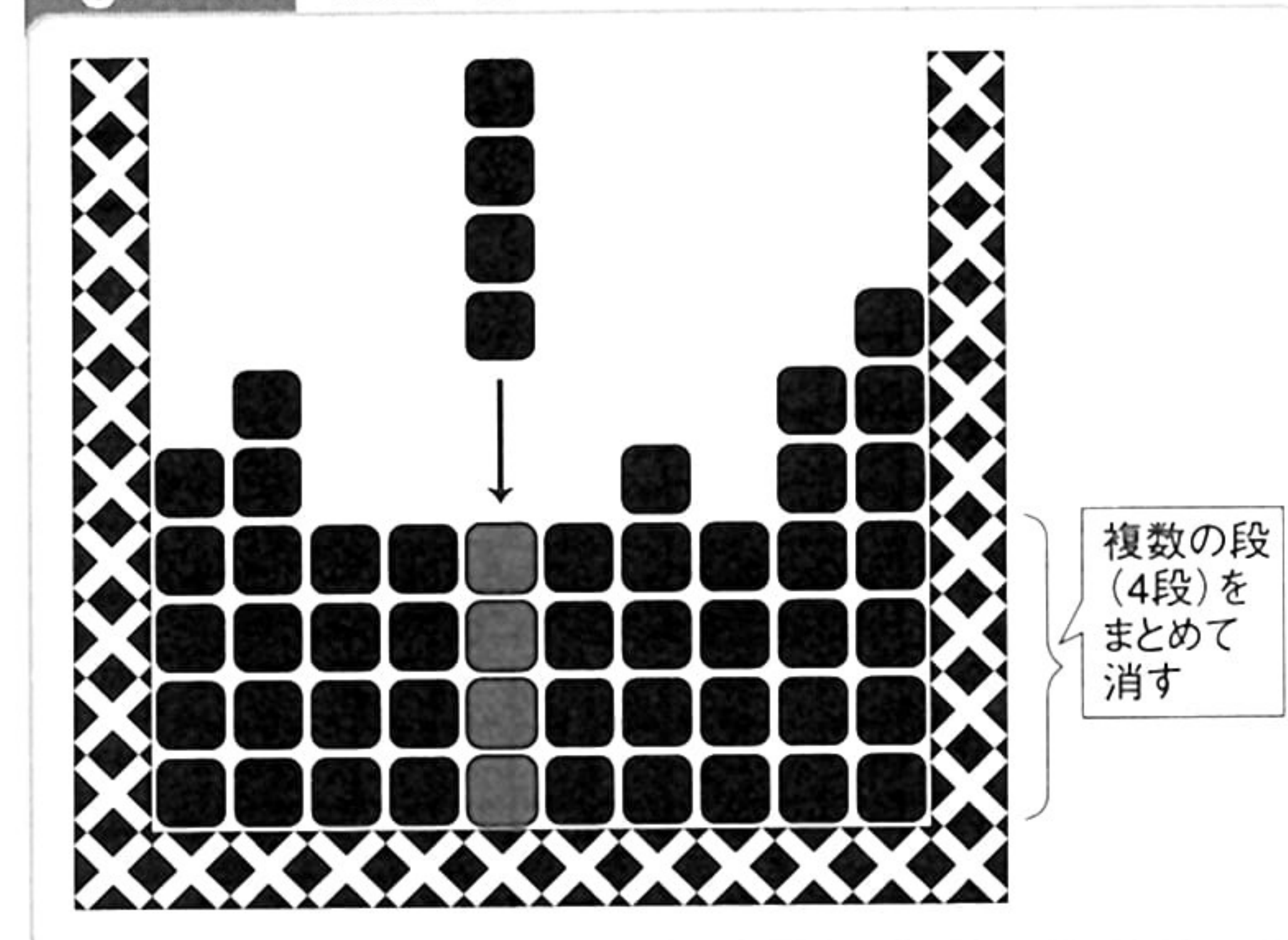



Fig. 2-25 消える段の間に消えない段がある例

消える段の間に
消えない段が
あってもよい

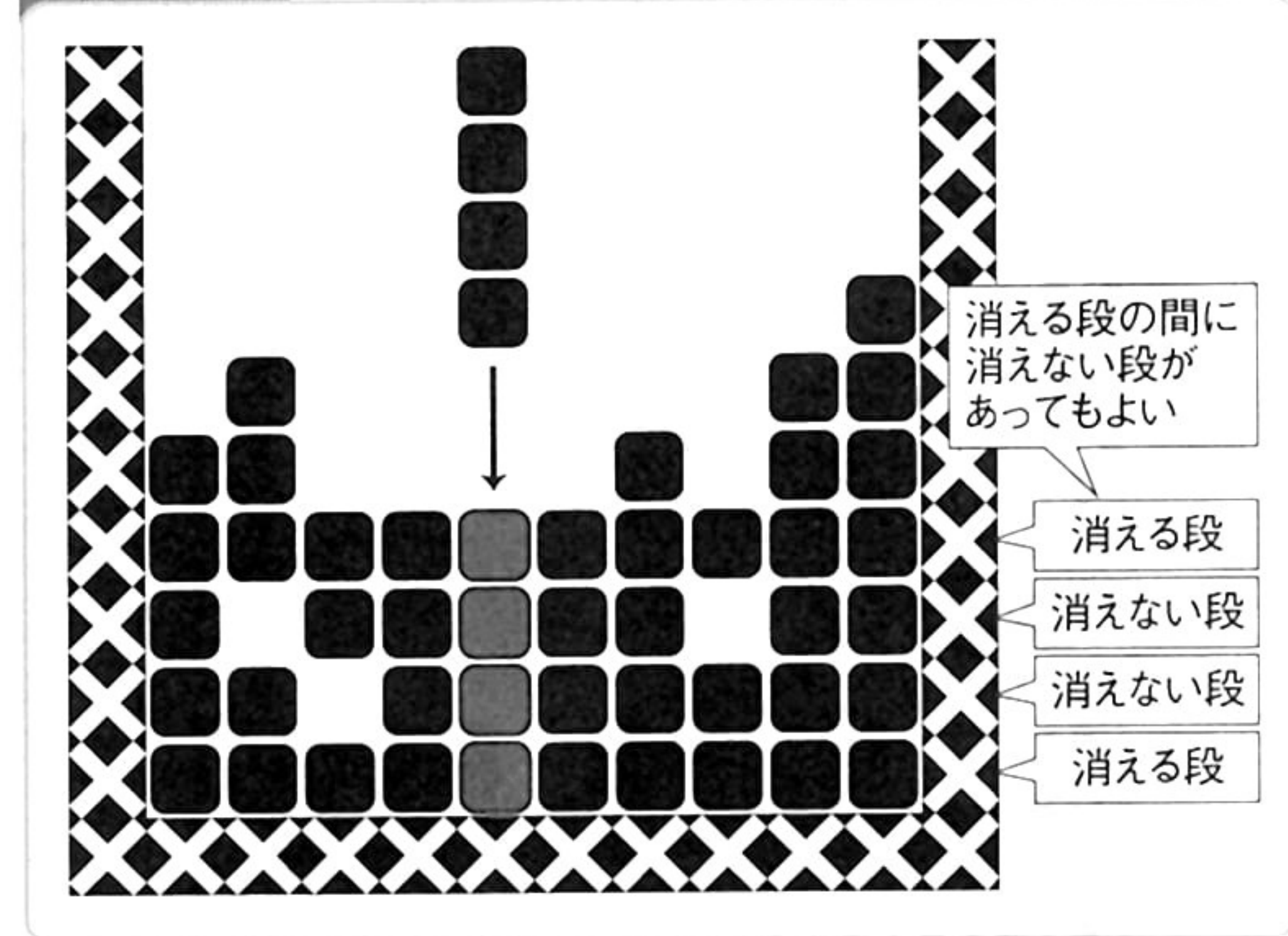
消える段

消えない段

消えない段

消える段

アルゴリズム

ブロックを消すには、ブロックが着地したときに、ステージのセルを調べます。ブロックが1列揃っている段、つまり空のセルがまったくない段があったら、その段は消すことができます (Fig. 2-26)。

消える段が見つかったら、その段にマークを付けます。ここでは、ブロックのセルを表す文字「#」を、文字「+」に変更して、消えるブロックを表すことにしました (Fig. 2-27)。

これから消えるブロックのセルを、すぐに空のセルに変更しないのは、ブロックが消えるアニメーションを表示するためです。文字「#」は普通のブロック、文字「+」は消えつつあるブロック、そして空白文字はブロックが完全に消えた場所、という具合にセルの状態を区別しておき、それぞれ別々のグラフィックを描画します。消えつつあるブロックに関しては、タイマーを使って、一定の時間をかけて少しずつブロックが薄くなっていく様子を表現します (Fig. 2-28)。

Fig. 2-26 消える段を探す

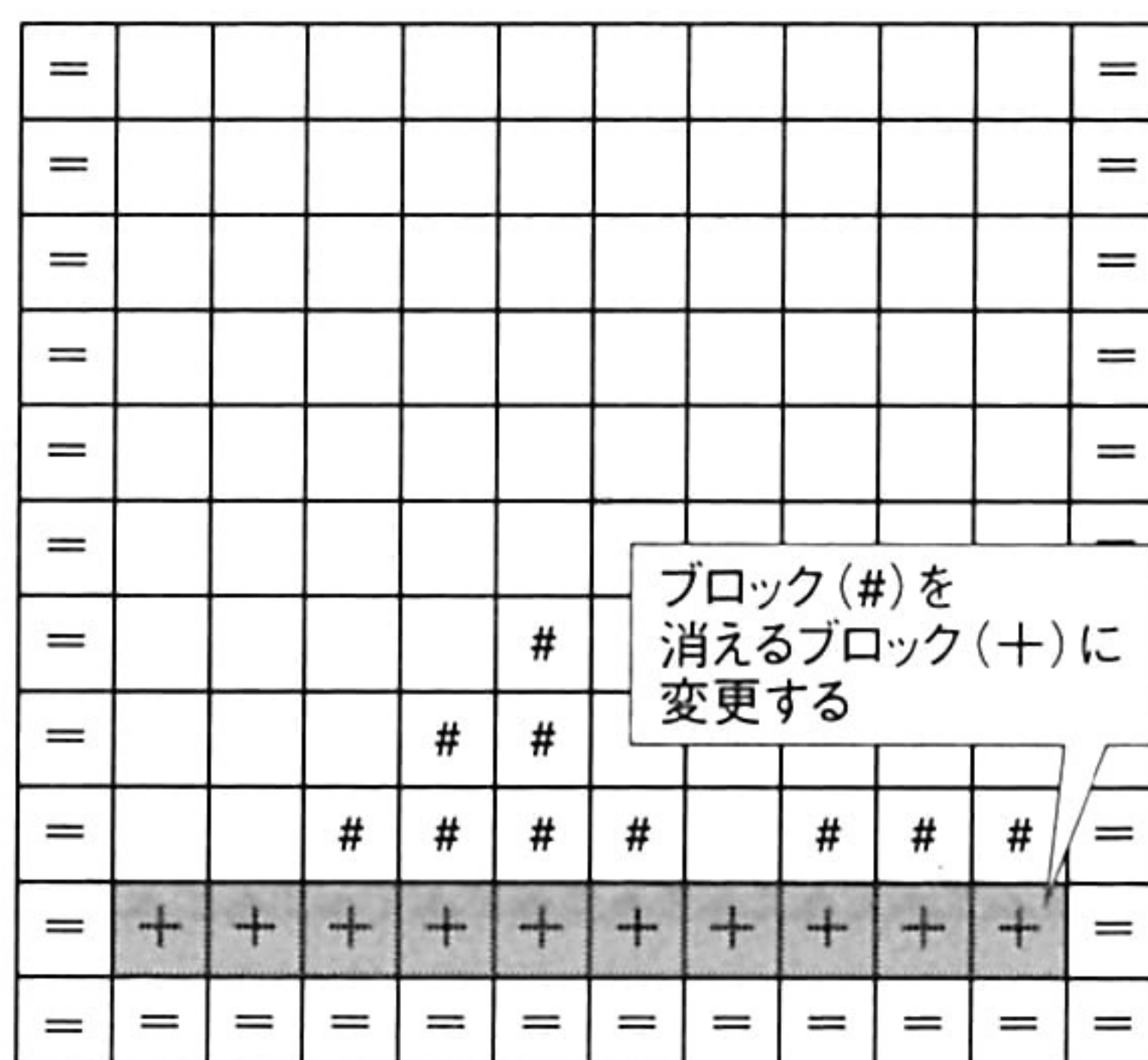
空のセルが
まったくない段は
消すことができる

Fig. 2-27 消えるブロックをマークする

ブロック(#)を
消えるブロック(+)に
変更する

Fig. 2-28 消えるブロックの表示

ステージの壁
(=のセル)
消えるブロック(+のセル)は
時間とともに薄くなるように描く
普通のブロック
(#のセル)

Fig. 2-29 上にあるブロックを落とす

消えた段よりも上にあるセルを
消えた段数分だけ下に移動させる

一定時間が経過したら、消えつつあるブロックを完全に消し、消えた段よりも上にあるブロックを落とします。そのためには、消えた段よりも上にあるセルを、1段ずつ下に移動させます（Fig. 2-29）。一度に複数の段を消した場合には、段数に応じてセルを下に移動させます。

プログラム

List 2-4はブロックを1段揃えて消すプログラムです。ブロックの移動処理と描画処理を掲載しました。ブロックの移動処理は、ブロックが着地したときの処理と、1段揃ったブロックを消すときの処理、そして新しいブロックを出現させる処理に分かれています。ここではそれぞれを、着地状態・消去状態・再出現状態と呼ぶことにします。

着地状態では、ステージのセルを調べて、ブロックが揃っている段を探します。揃った段が見つかったら、ブロックのセル（#）を、消えるブロックのセル（+）に変化させます。そしてタイマーを設定して、消去状態に移行します。もしも揃った段がなければ、再出現状態に移行します。

消去状態では、タイマーを更新します。タイマーが一定値に達したら、ステージのセルを調べて、消えるブロックのセル（+）を探します。セルが見つかったら、その段よりも上にあるセルを落下させます。最後に、再出現状態に移行します。

再出現状態では、新しいブロックを画面上方に出現させます。具体的な処理は「次のブロックを表示する」（→p. 70）で解説します。

ブロックの描画処理では、ステージのセルを調べて、ブロックと壁を描画します。消えるブロック（+）に関しては、タイマーを使って、だんだん表示が薄くなって消えるように、色を変化させながら描画します。

List 2-4 ブロックを1段揃えて消す（CDroppingBlockBlockクラス）

```
// ブロックの移動処理
bool CDroppingBlockBlock::Move(const CInputState* is) {

    // …（中略）…

    // 着地状態の処理
    if (State==2) {

        // 再出現状態に移行する
        // 消えるブロックがある場合には、
        // 消去状態に移行するよう変更する
        State=4;

        // ブロックが隙間なく揃った段を探す
        for (
            int y=DROPPING_BLOCK_FIELD_TOP;
```

```
            y<DROPPING_BLOCK_FIELD_BOTTOM;
            y++
        ) {
            int x;
            for (
                x=DROPPING_BLOCK_FIELD_LEFT;
                x<DROPPING_BLOCK_FIELD_RIGHT;
                x++
            ) {
                // 空のセルがある場合にはループを抜け出す
                if (StageCell->Get(x, y)==' ') break;
            }

            // ブロックが揃った段がある場合の処理
            if (x>=DROPPING_BLOCK_FIELD_RIGHT) {
                for (
                    x=DROPPING_BLOCK_FIELD_LEFT;
                    x<DROPPING_BLOCK_FIELD_RIGHT;
                    x++
                ) {
                    // 段のブロックを消えるブロック（＋）にする
                    StageCell->Set(x, y, '+');
                }

                // 消去状態に移行する
                Time=0;
                State=3;
            }
        }
    }

    // 消去状態の処理
    if (State==3) {

        // タイマーの更新
        Time++;

        // タイマーが一定値に達したら、ブロックを消す
        if (Time==20) {

            // 消えるブロック（＋）がある段を探す
            for (
                int i=DROPPING_BLOCK_FIELD_TOP;
                i<DROPPING_BLOCK_FIELD_BOTTOM;
                i++
            ) {
                // 消えるブロックが見つかったときの処理
                if (
                    StageCell->Get(
```

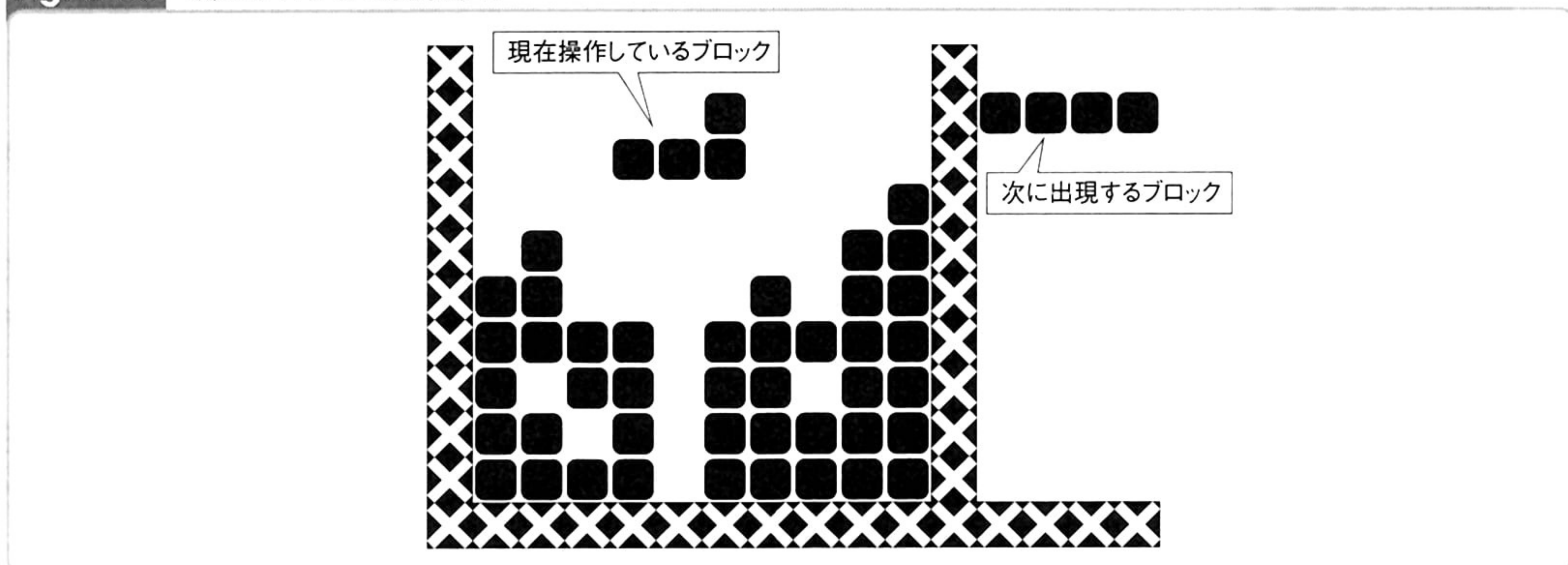
```
                        DROPPING_BLOCK_FIELD_LEFT, i)=='+'
                ) {
                    //消える段よりも上にあるすべての段を、
                    //1段ずつ下に落とす
                    for (
                        int y=i;
                        y>=DROPPING_BLOCK_FIELD_TOP;
                        y--
                    ) {
                        for (
                            int x=DROPPING_BLOCK_FIELD_LEFT;
                            x<DROPPING_BLOCK_FIELD_RIGHT;
                            x++
                        ) {
                            StageCell->Set(x, y, StageCell->Get(x, y-1));
                        }
                    }
                }
            }

            //再出現状態に移行する
            State=4;
        }
    }

    //再出現状態の処理
    if (State==4) {

        //新しいブロックを出現させる
        Init();
    }

    return true;
}

//ブロックの描画処理
void CDroppingBlockBlock::Draw() {

    //画面の解像度を取得する
    float
        sw=Game->GetGraphics()->GetWidth()/MAX_X,
        sh=Game->GetGraphics()->GetHeight()/MAX_Y;

    //...(中略)...

    //タイマーの値に応じて色を決める
    float f=Time*0.05f;

    //ステージのすべてのセルについて処理する
```

```
    for (int y=0; y<MAX_Y; y++) {
        for (int x=0; x<MAX_X; x++) {

            // セルの種類によって分岐する
            switch (StageCell->Get(x, y)) {

                // ブロックのセル(#)を描画する
                case '#':
                    Game->Texture[TEX_OBJECT]->Draw(
                        x*sw, y*sh, sw, sh,
                        0, 0, 1, 1, COL_BLACK
                    );
                    break;

                // 消えるブロックのセル(+)は、
                // タイマーの値に応じた色で描画する
                case '+':
                    Game->Texture[TEX_OBJECT]->Draw(
                        x*sw, y*sh, sw, sh,
                        0, 0, 1, 1, D3DXCOLOR(f, f, f, 1)
                    );
                    break;

                // 壁のセル(=)を描画する
                case '=':
                    Game->Texture[TEX_DROP_FLOOR]->Draw(
                        x*sw, y*sh, sw, sh,
                        0, 0, 1, 1, COL_BLACK
                    );
                    break;
            }
        }
    }
}
```

次のブロックを表示する

現在操作しているブロックの、次に出現するブロックを表示する機能です。この機能によって、次に落ちてくるブロックを見ながら戦略を立てることができるため、より効率よくブロックを積んでいくことができます。

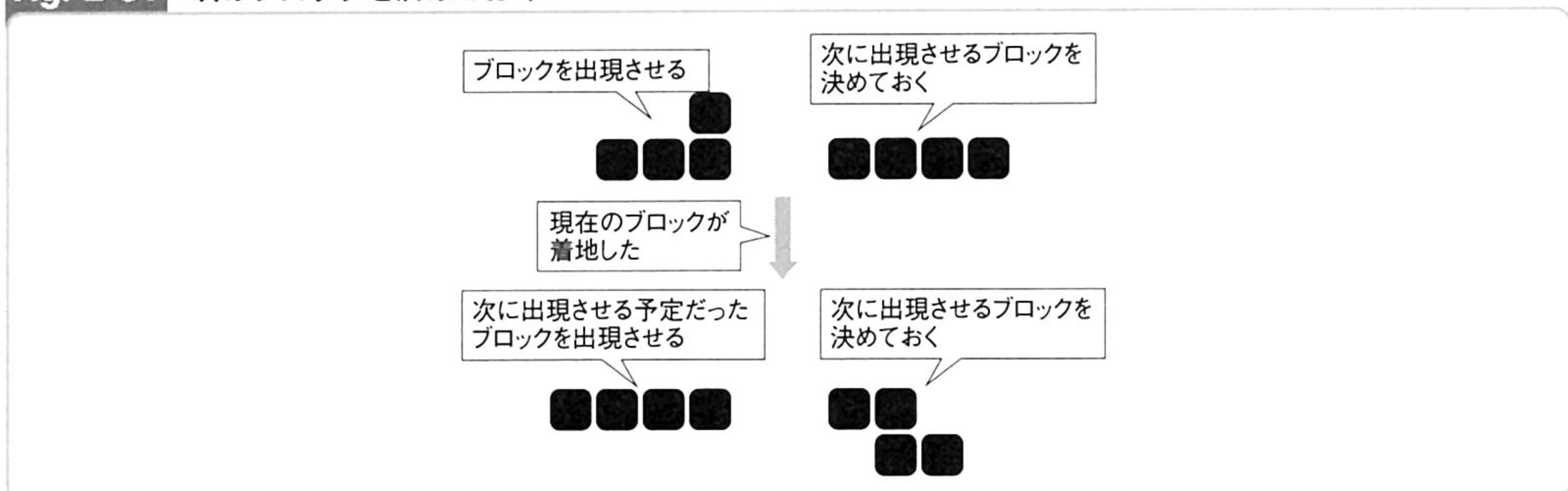
次のブロックは画面端などに表示されます(Fig. 2-30)。プレイヤーは次のブロックを横目で見ながら、現在落下しているブロックを操作します。

Fig. 2-30　次のブロックを表示する

次のブロックを表示する機能は、多くのゲームに採用されています。『テトリス』『コラムス』『ぷよぷよ』『対戦ぱずるだま』など、ほとんどの落ち物パズルゲームは、次に出現するブロックやボールを表示しています。

アルゴリズム

次のブロックを表示するには、現在のブロックを出現させるときに、次にどんなブロックを出現させるのかを一緒に決めておきます（Fig. 2-31）。そして、現在のブロックをステージに表示し、次のブロックは画面端などに表示します。

現在のブロックが着地したら、決めておいた次のブロックを出現させます。そして同時に、新しい次のブロックを決めます。このように、現在のブロックを出現させるときに、必ず次のブロックも決めておくことがポイントです。

Fig. 2-31　次のブロックを決めておく

プログラム

List 2-5は次のブロックを表示するプログラムです。ブロックの初期化処理と、ブロックの描画処理を掲載しました。

初期化処理では、決めておいた次のブロックを現在のブロックにします。そして、新しい次のブロックを決めます。ここでは乱数を使って、7種類あるブロックのなかから1種類を選んでいます。

描画処理については、次のブロックを描画する処理を抜粋しました。次のブロックは、現在のブロックと同じ方法で描画することができます。違いは、ステージ内ではなく、画面端などに表示することです。

List 2-5 次のブロックを表示する(CDroppingBlockBlockクラス)

```
// ブロックの初期化処理
void CDroppingBlockBlock::Init() {

    // 決めておいた次のブロックを現在のブロックにする
    CurrentBlock=NextBlock;

    // 乱数を使って次のブロックを決める
    NextBlock=BlockCell+Rand.Int31()%
        DROPPING_BLOCK_PATTERN_COUNT*
        DROPPING_BLOCK_TURN_COUNT;

    // 現在のブロックのセル座標を設定する
    CX=(DROPPING_BLOCK_FIELD_LEFT+
        DROPPING_BLOCK_FIELD_RIGHT-
        DROPPING_BLOCK_CELL_SIZE)/2;
    CY=DROPPING_BLOCK_FIELD_TOP;

    // 描画座標の設定
    X=CX;
    Y=CY;

    // 状態・落下タイマー・回転角度の設定
    State=0;
    DropTime=0;
    Turn=0;

    // 現在のブロックが出現する位置に、
    // 他のブロックがある場合には、
    // ゲームオーバーにする
    if (StageCell->Hit(CX, CY, CurrentBlock[Turn])) {

        // このサンプルでは省略しているが、
```

```
            // 本来はゲームオーバー時の処理を記述する
            State=99999;
        }
    }

    // ブロックの描画処理
    void CDroppingBlockBlock::Draw() {

        // 画面の解像度を取得する
        float
            sw=Game->GetGraphics()->GetWidth()/MAX_X,
            sh=Game->GetGraphics()->GetHeight()/MAX_Y;

        // ...(中略)...

        // ブロックを描画する
        for (int y=0; y<DROPPING_BLOCK_CELL_SIZE; y++) {
            for (int x=0; x<DROPPING_BLOCK_CELL_SIZE; x++) {

                // ...(中略)...

                // 次のブロックを描画する
                if (NextBlock[0]->Get(x, y)=='#') {
                    Texture->Draw(
                        (DROPPING_BLOCK_NEXT_LEFT+x)*sw,
                        (1+y)*sh, sw, sh, 0, 0, 1, 1, COL_BLACK
                    );
                }
            }
        }
        // ...(中略)...
    }
```

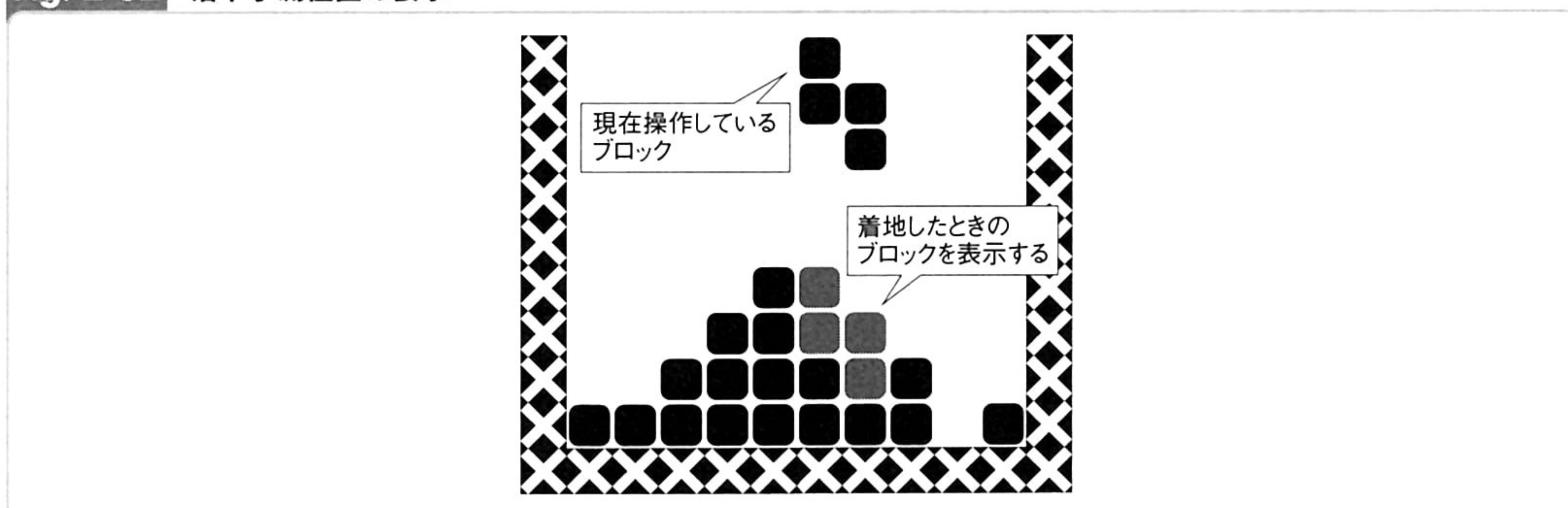
落下予測位置を表示する

ブロックを真下へ落としたときに、どこに着地するのかを表示する機能です。ブロックを隙間なく詰めるには、ブロックをどこに落とすのかが重要です。こういったゲームでは、プレイヤーは常に落下位置を予測しながらブロックを操作します。落下予測位置の表示機能があると、一目でブロックの行き先がわかり、より素早い操作が可能になります。

落下予測位置には、着地したときのブロックの様子が表示されます (Fig. 2-32)。操作しているブロックと区別するために、落下予測位置は薄く表示したり、異なる色で表示したりするとよいでしょう。

Fig. 2-32 落下予測位置の表示

現在操作しているブロック

着地したときのブロックを表示する

落下予測位置の表示は『テトリス ザ・グランドマスター』に採用されています。落下予測位置の表示がない『テトリス』に比べると、ブロックが落ちる位置が一目でわかるため、よりスピーディーなゲーム展開が楽しめます。

アルゴリズム

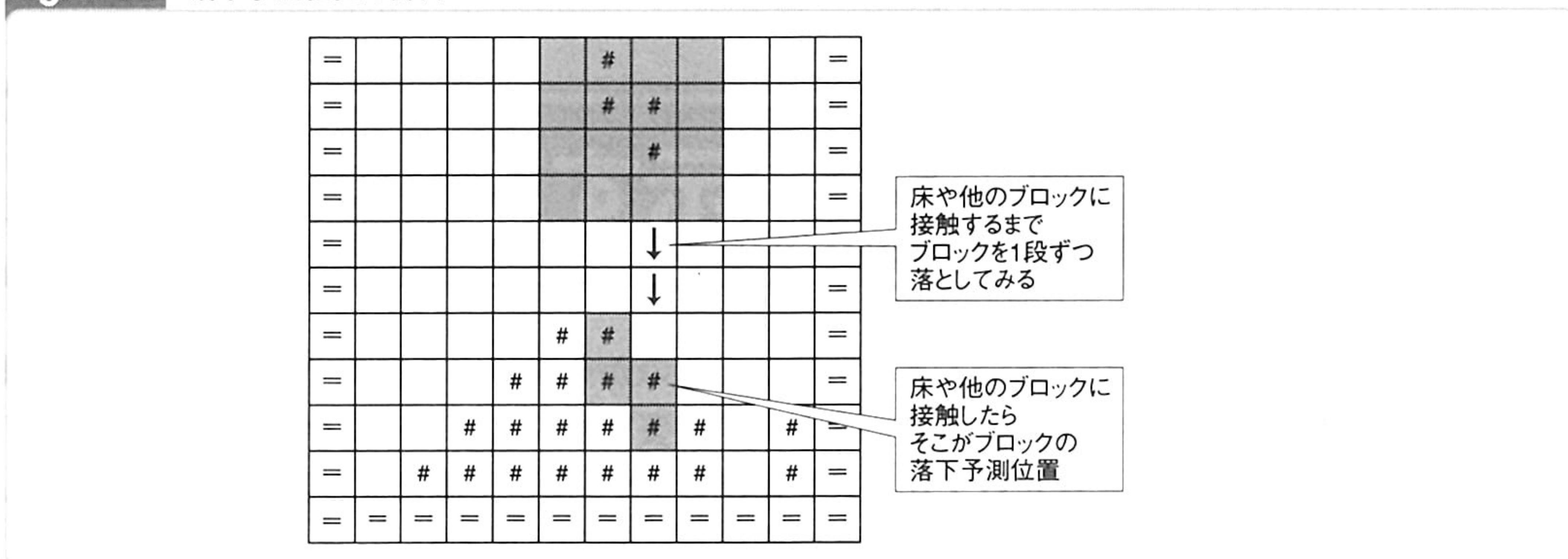
落下予測位置を表示するには、ブロックを現在いる位置から1段ずつ落としてみます。もちろん、画面上ではブロックを動かさずに、座標の計算だけを行います(Fig. 2-33)。

ブロックのセルとステージのセルとの間で当たり判定処理を行い、ブロックが床や他のブロックに接触するまで、ブロックを落とし続けます。接触したら、そこがブロックの落下予測位置です。あとは、この位置にブロックを薄く表示します。

Fig. 2-33 落下予測位置の計算

=						#					=
=						#	#				=
=							#				=
=											=
=							↓				
=							↓				=
=					#	#					=
=				#	#	#	#				=
=			#	#	#	#	#	#		#	=
=		#	#	#	#	#	#	#		#	=
=	=	=	=	=	=	=	=	=	=	=	=

床や他のブロックに接触するまでブロックを1段ずつ落としてみる

床や他のブロックに接触したらそこがブロックの落下予測位置

プログラム

List 2-6は落下予測位置を表示するプログラムです。このサンプルでは、ブロックの描画処理において、落下予測位置の計算と表示を行います。

落下位置を計算するには、ブロックのY座標を加算しつつ、ブロックとステージの当たり判定処理を行います。接触したら、その位置が落下予測位置です。このサンプルでは、現在位置にはブロックを黒色で描き、落下予測位置には灰色で描いています。

List 2-6　落下予測位置を表示する（CDroppingBlockBlockクラス）

```
// ブロックの描画処理
void CDroppingBlockBlock::Draw() {

	// 画面の解像度を取得する
	float
		sw=Game->GetGraphics()->GetWidth()/MAX_X,
		sh=Game->GetGraphics()->GetHeight()/MAX_Y;

	// 落下予測位置の計算
	// ブロックが床や他のブロックに接触するまで、
	// Y座標を加算していく
	int ey;
	for (ey=CY; ey<DROPPING_BLOCK_FIELD_BOTTOM; ey++) {

		// 接触したら、そこが落下予測位置
		if (StageCell->Hit(CX, ey+1, CurrentBlock[Turn])) break;
	}

	// ブロックの描画
	for (int y=0; y<DROPPING_BLOCK_CELL_SIZE; y++) {
		for (int x=0; x<DROPPING_BLOCK_CELL_SIZE; x++) {
			if (State<2 && CurrentBlock[Turn]->Get(x, y)=='#') {

				// 落下予測位置に灰色でブロックを描画する
				Texture->Draw(
					(CX+x)*sw, (ey+y)*sh,
					sw, sh, 0, 0, 1, 1, COL_LGRAY
				);

				// 現在位置に黒色でブロックを描画する
				Texture->Draw(
					(X+x)*sw, (Y+y)*sh,
					sw, sh, 0, 0, 1, 1, COL_BLACK
				);
			}
			// …（中略）…
```

```
            }
        }
        // ...（中略）...
    }
```

宝石を落とす

宝石を落として積み上げていくアクションです。ボタンを入力すると、落ちてくる宝石の並び順が変わります。同じ種類の宝石を縦横斜めに並べて積むと、宝石を消すことができます。

落ちてくる宝石は、いくつかの宝石が縦に並んだものです（Fig. 2-34）。ここでは3個の宝石が並んでいる場合を考えます。

宝石の種類はランダムに決まります。異なる宝石の組み合わせが落ちてくることも、同じ宝石の組み合わせが落ちてくることもあります。

レバーを左右に入力すると、宝石を左右に動かすことができます。また、宝石は時間とともに少しずつ落下します。レバーを下に入力すると、速く落下させることが可能です。

ステージの床に落ちるか、あるいは積み上げられた他の宝石の上に落ちると、宝石は着地します（Fig. 2-35）。着地の後、宝石の種類が縦横斜めに揃うと、揃った部分の宝石が消えます。

宝石を落として積み上げるゲームとしては、『コラムス』がよく知られています。物体を落として積み上げるという操作は『テトリス』に似ていますが、宝石の順番を変えたり、縦横斜めに宝石を並べたりといったルールが異なるので、まったく違ったゲームに仕上がっています。どちらのゲームも目的は同じで、ブロックや宝石を上手にさばいて、次々に消していくことです。

Fig. 2-34 縦に並んだ宝石

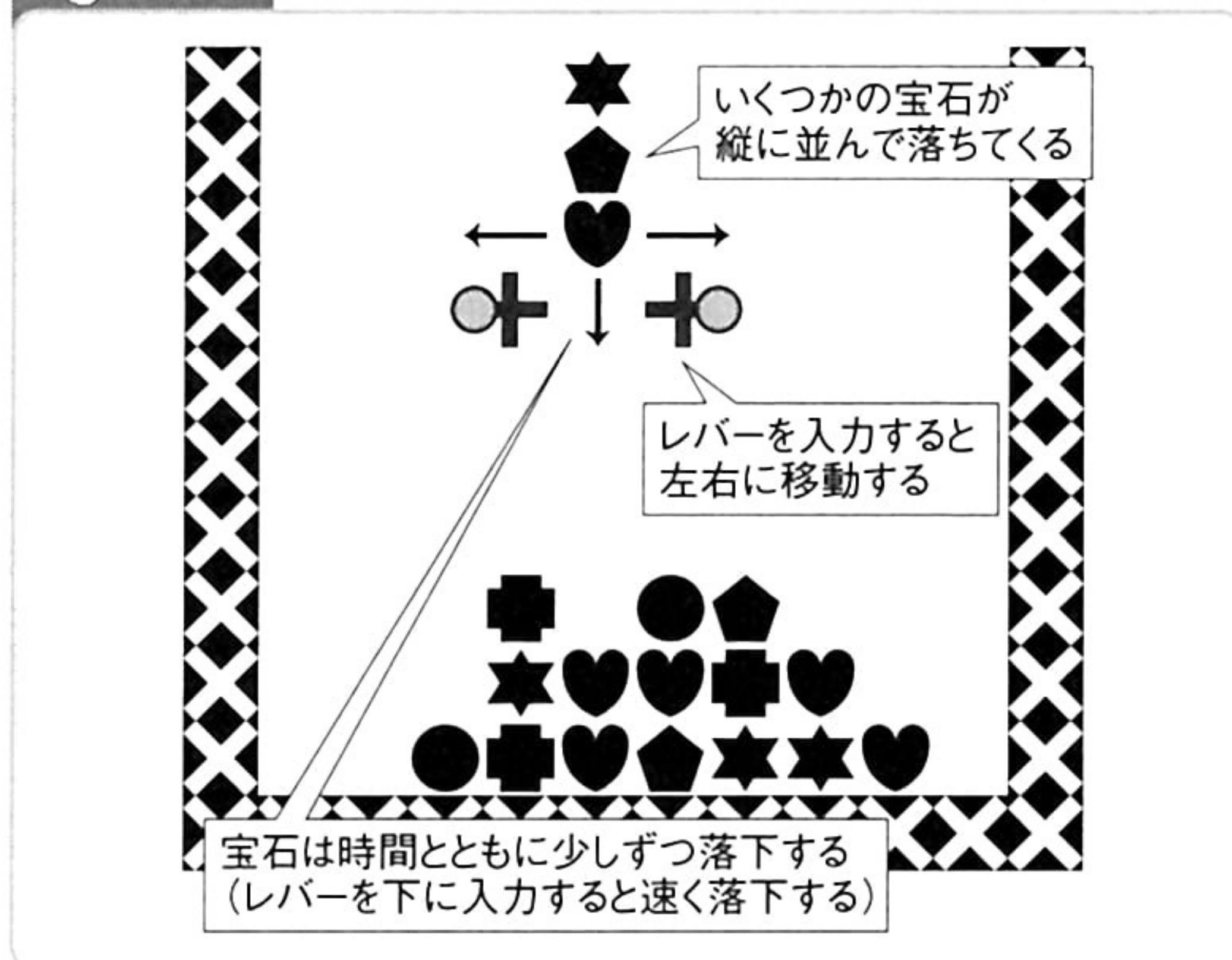
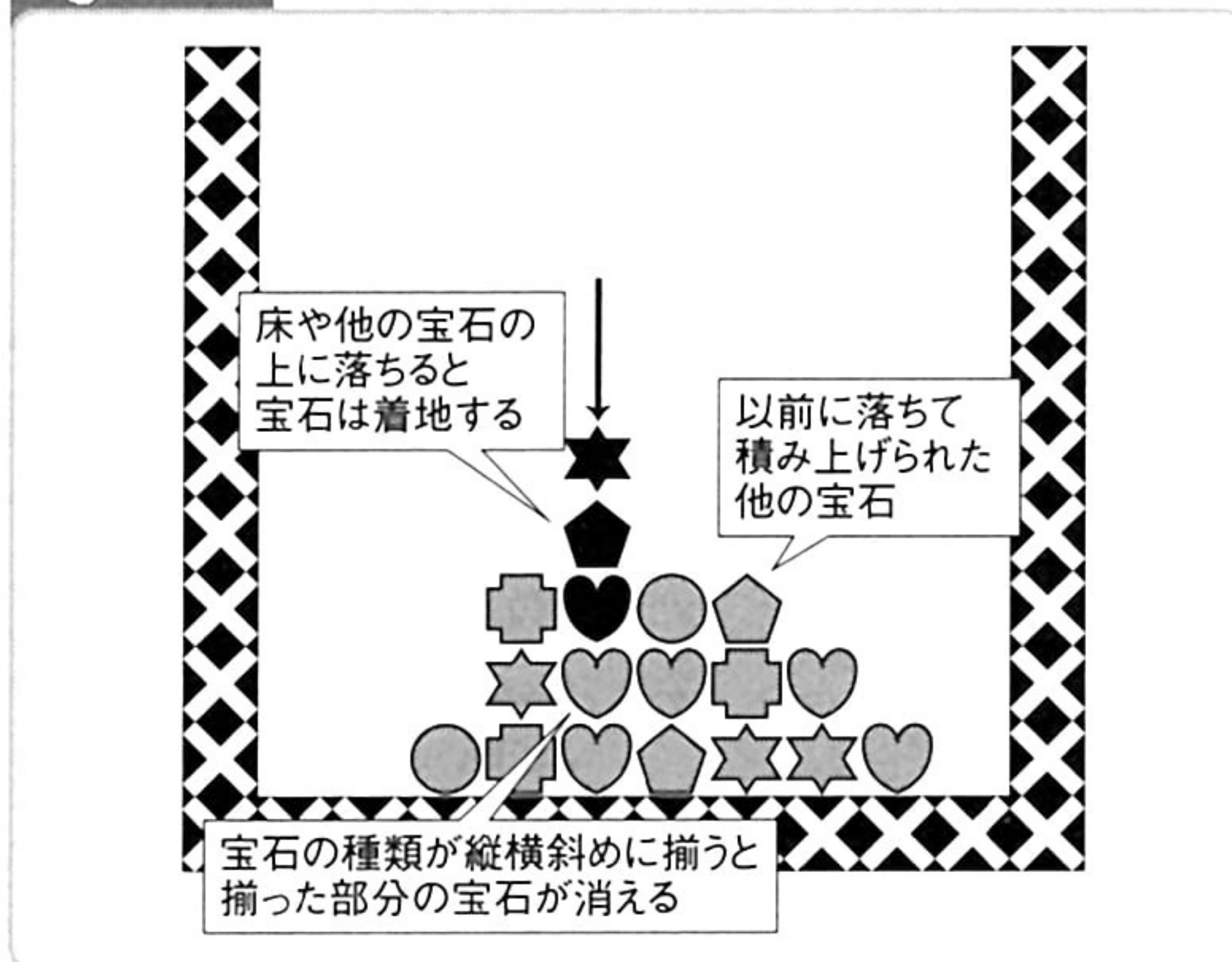



Fig. 2-35 宝石の着地

アルゴリズム

宝石を落とすアクションは、「ブロックを落とす」(→p. 50) と同じ方法で実現できます。左右への移動は「ブロックを左右に移動する」(→p. 57) と同様です。

落ちてくる宝石はセルで表現します (Fig. 2-36)。ここでは「1×3」のセルを使いましょう。「0〜4」の数字で宝石の種類を表します。

ステージもセルで表現します (Fig. 2-37)。ステージの周囲を囲む壁は文字「＝」で表し、積み上げられた宝石は「0〜4」の数字で表します。

宝石を時間とともに落下させるには、タイマーを使います (Fig. 2-38)。タイマーを更新し、タイマーが一定値になるたびに、宝石のセル座標を更新します。

レバーを下に入力したときには、タイマーが一定値になるのを待たずに、すぐにセル座標を更新します。これで、レバー入力があったときには宝石の落下スピードが上がります。

セル座標を更新する前に、宝石のセルとステージのセルとの間で、当たり判定処理を行います (Fig. 2-39)。宝石がステージの床や他の宝石に接触しなければ、宝石を落とします。

Fig. 2-36 宝石をセルで表現する

宝石を数字 (0〜4) のセルで表現する
落ちてくる宝石を1×3のセルで表現する

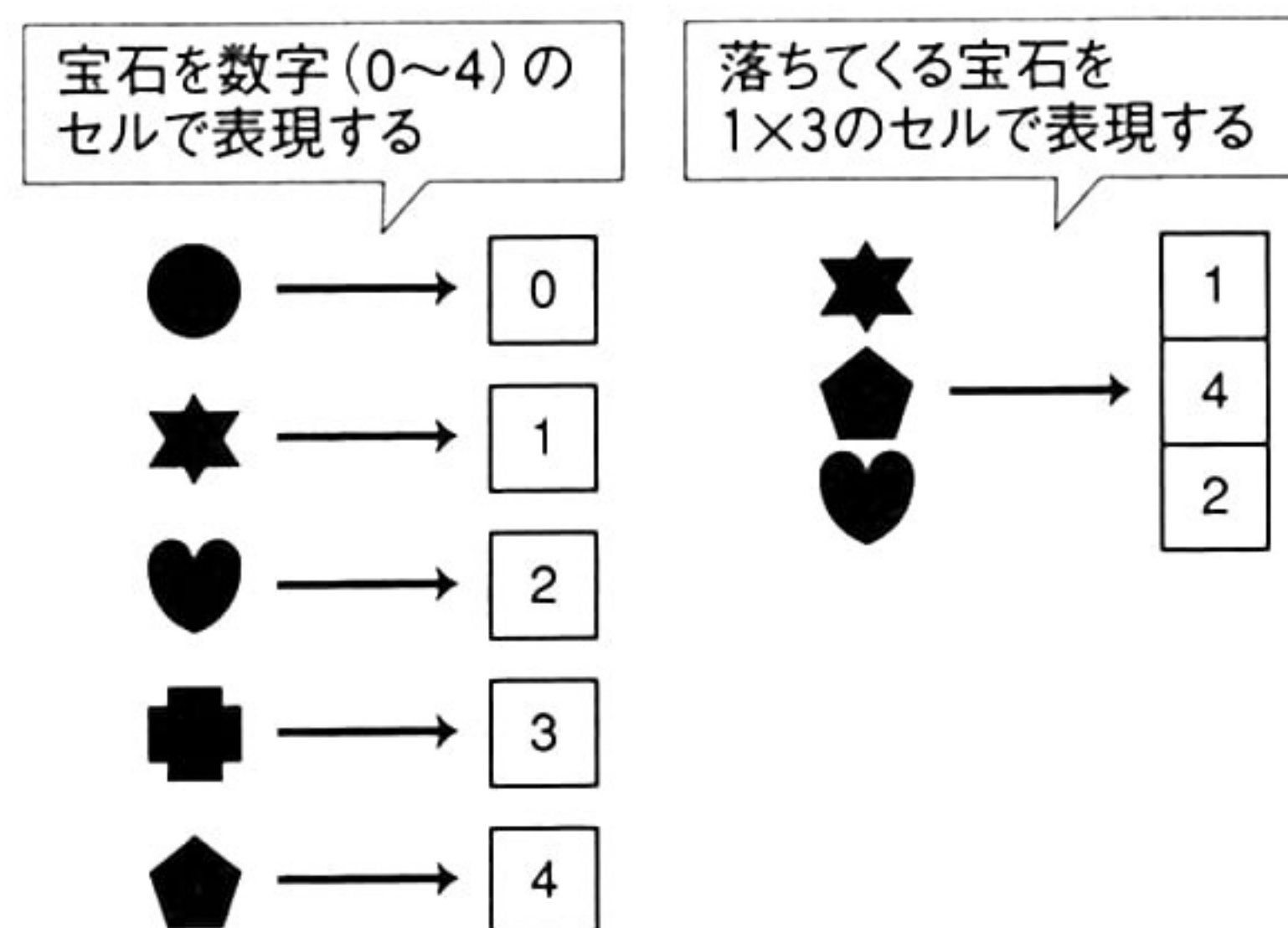
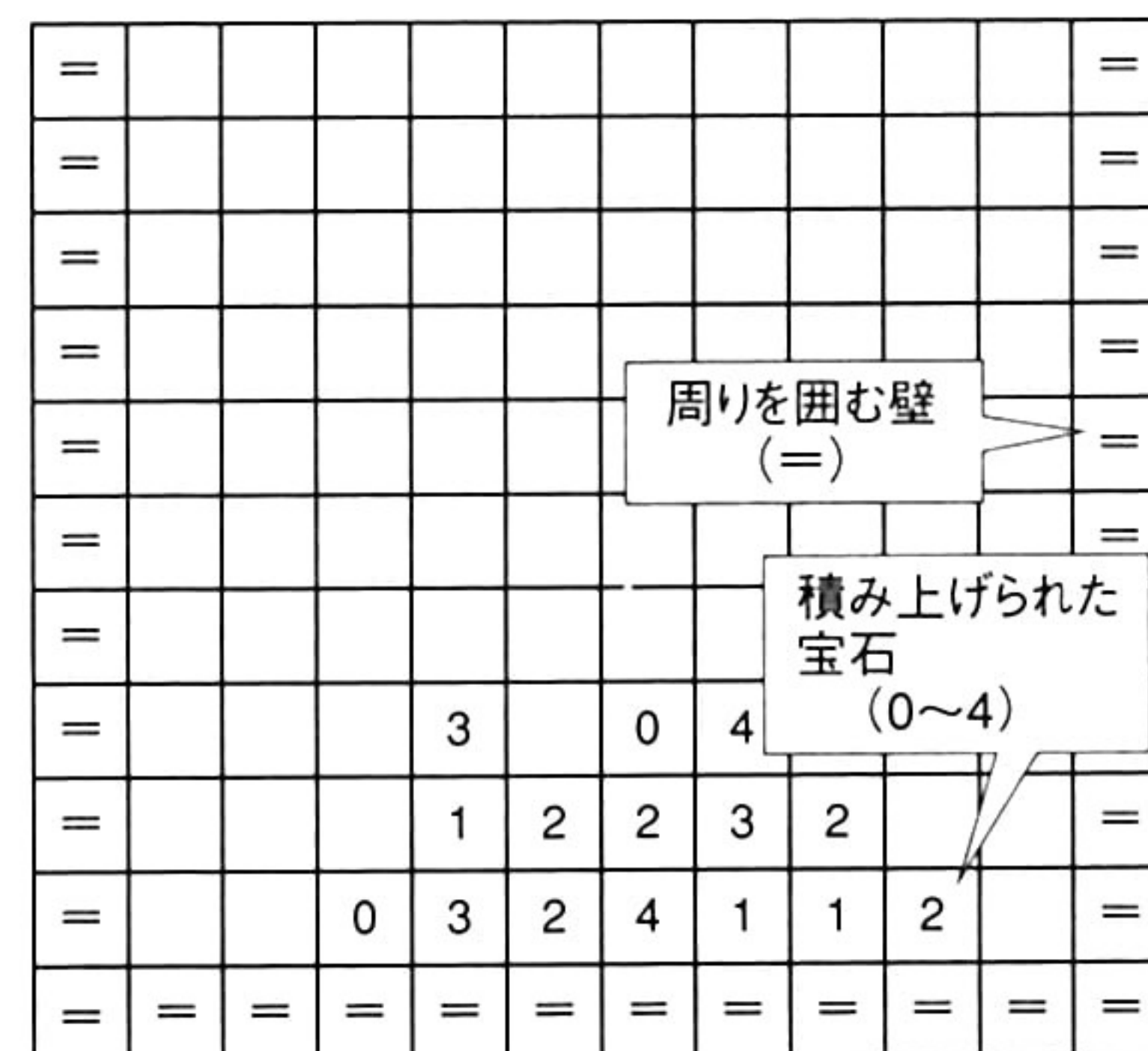



Fig. 2-37 ステージをセルで表現する

周りを囲む壁 (＝)
積み上げられた宝石 (0〜4)

Fig. 2-38 時間とともに宝石を落下させる

セル座標 (CX, CY)
一定時間が経過する (タイマーを使用)
セル座標を更新 (CX, CY+1)
レバーを下に入力したときにはタイマーを待たずにすぐにセル座標を更新する

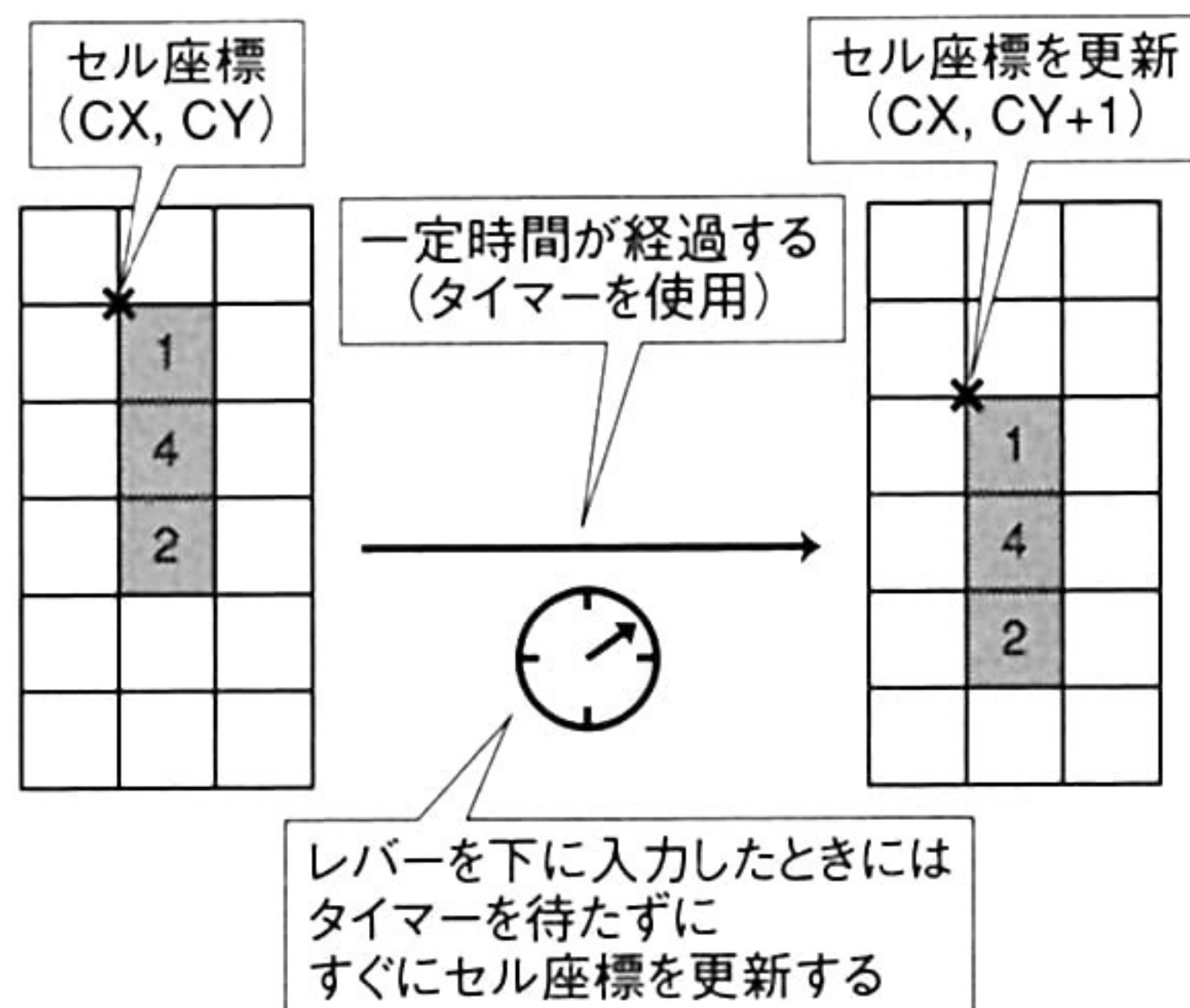



Fig. 2-39 宝石とステージの当たり判定処理

宝石のセル (1×3)
宝石のセルとステージの空ではないセルとの接触を調べる
接触しないかぎりは宝石を落下させることができる

Fig. 2-40 宝石の着地とセルの合成

宝石が床や他の宝石に接触して落下させることができなくなったら着地させる

Fig. 2-41 左右への移動

宝石のセルとステージの空ではないセルとの接触を調べる

接触しないかぎりは宝石を左右に移動できる

接触する場合には、宝石を着地させます（Fig. 2-40）。宝石のセルはステージのセルに合成します。

レバーを左右に入力したときには、宝石を左右に移動します。宝石のセルとステージのセルとの間で当たり判定処理を行い、接触しなければ、宝石のセル座標を更新します（Fig. 2-41）。接触する場合には、宝石を移動させません。

プログラム

List 2-7は宝石を落とすプログラムです。宝石の移動処理を掲載しました。

移動処理は入力状態・移動状態に分かれています。入力状態では、落下タイマーを使って宝石を少しずつ落下させます。また、レバー入力に応じて宝石を速く落下させたり、左右に移動させたりする処理も行います。

宝石を落下させるときと、左右に移動させるときには、移動状態に移行します。移動状態では、宝石の描画座標を少しずつ変化させることによって、画面上で宝石が滑らかに動くようにします。これは「ブロックを落とす」（→p. 50）と同様です。

List 2-7 宝石を落とす（CDroppingJewelJewelクラス）

```
// 宝石の移動処理
bool CDroppingJewelJewel::Move(const CInputState* is) {

    // 入力状態
    if (State==0) {

        // 落下タイマーの更新
        DropTime++;

        // レバーを下に入力するか、
```

```
//落下タイマーが一定値に達したら、
//宝石を落下させる
if ((is->Down && !PrevDown)|| DropTime==60) {

    //ステージに接触したら、宝石を着地させる
    if (StageCell->Hit(CX, CY+1, CurrentJewel)) {

        //ステージのセルに宝石のセルを合成する
        StageCell->Merge(CX, CY, CurrentJewel);

        //着地状態に移行する
        State=2;
    } else

    //ステージに接触しない場合には、
    //宝石を1段落下させる
    {
        //落下タイマーの設定
        DropTime=0;

        //セル座標の更新
        CY++;

        //速度・タイマーの設定
        VX=0;
        VY=1;
        Time=0;

        //移動状態に移行する
        State=1;
    }
} else

//レバーを左に入力しており、
//かつステージに接触しないならば、
//左に移動する
if (is->Left && !StageCell->Hit(CX-1, CY, CurrentJewel)) {

    //セル座標の更新
    CX--;

    //速度・タイマーの設定
    VX=-1;
    VY=0;
    Time=0;

    //移動状態に移行する
    State=1;
} else
```

```
		// レバーを右に入力しており、
		// かつステージに接触しないならば、
		// 右に移動する
		if (is->Right && !StageCell->Hit(CX+1, CY, CurrentJewel)) {

		// セル座標の更新
			CX++;

			// 速度・タイマーの設定
			VX=1;
			VY=0;
			Time=0;

			// 移動状態に移行する
			State=1;
		} else

		// ...(中略)...
	}

	// 移動状態
	if (State==1) {

		// タイマーの更新
		Time++;

		// 描画座標の更新
		X=CX-VX*(1-Time*0.1f);
		Y=CY-VY*(1-Time*0.1f);

		// タイマーが一定値に達したら、
		// 入力状態に移行する
		if (Time==10) {
			State=0;
		}
	}

	// ...(中略)...
}
```

SAMPLE

「DROPPING JEWEL」は「宝石を落とす」「宝石の順番を変える」「縦横斜めに揃える」のサンプルです。

レバーの左右(カーソルキーの左右)で宝石が移動し、レバーの下(カーソルキーの下)で宝石の落下スピードが上がります。ボタン0(Zキー)を押すと、宝石の並び順が変わります。

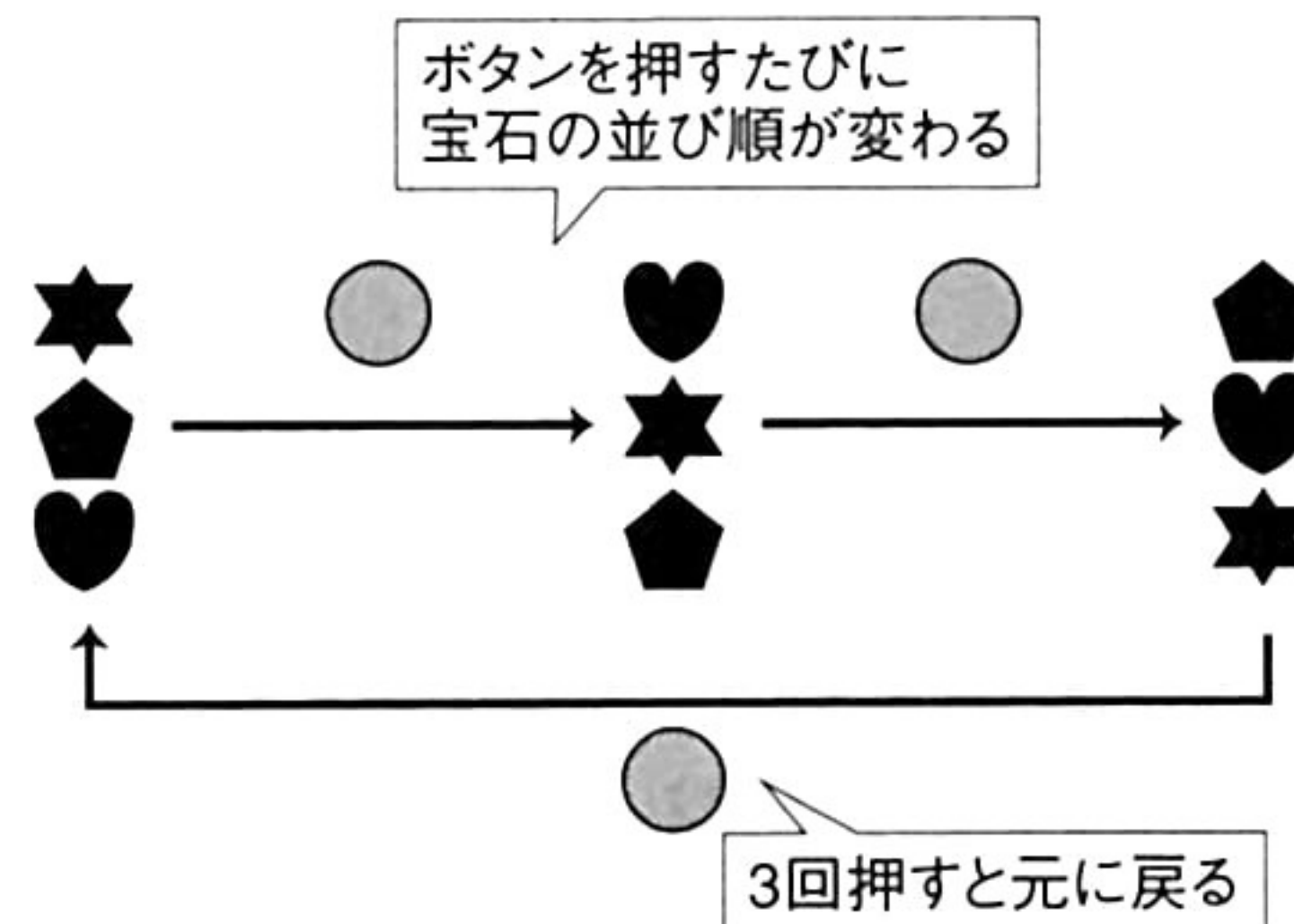
同じ種類の宝石を縦横斜めに3個以上並べると、宝石を消すことができます。宝石を消すと、上に載っていた宝石が落下して、再び縦横斜めに同じ種類が並び、連鎖的に消えることもあります。なお、画面右端に表示されている宝石は、次に落ちてくる宝石です。

DROPPING JEWEL → p. 386

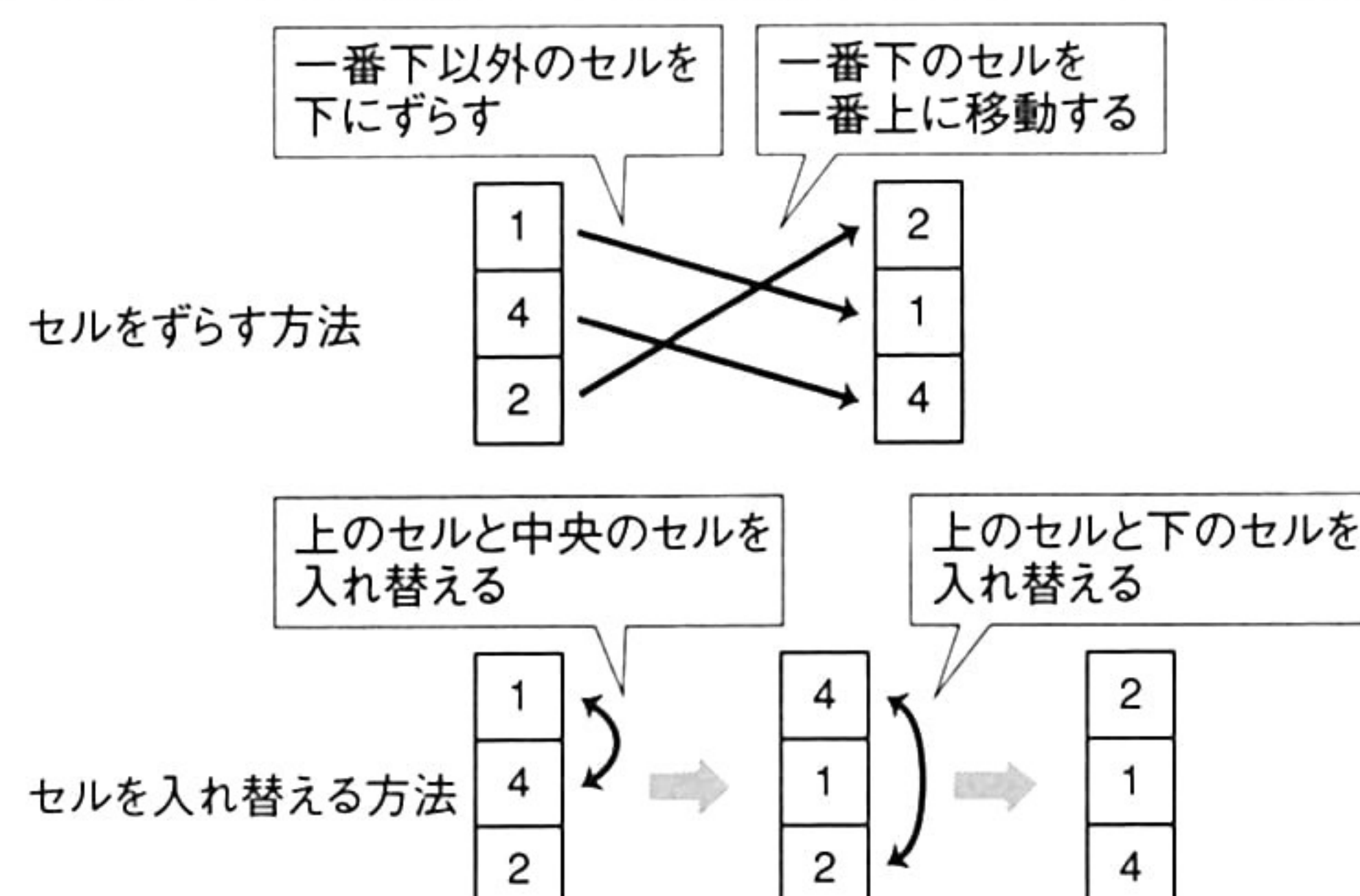
宝石の順番を変える

ボタン入力で宝石の並び順を変えるアクションです。順番を変えることによって、宝石を積み上げるときに、宝石の種類を揃えやすくなります。

ボタンを入力すると、宝石の並び順を変えることができます(Fig. 2-42)。順番を変えるたびに、上の宝石が中央へ、中央の宝石が下へ、下の宝石が上へ移動します。図のように宝石が3個の場合には、3回ボタンを押すと元の順番に戻ります。

Fig. 2-42　宝石の順番を変える

アルゴリズム

ボタンの入力を検出したら、宝石の順番を変えます。順番を変えるには、宝石のセルを操作します(Fig. 2-43)。一番下以外のセルを1個ずつ下にずらして、一番下のセルを一番上に移動します。あるいは、宝石のセルを入れ替えてもかまいません。上のセルと中央のセルを入れ替え、続いて上のセルと下のセルを入れ替えれば、順番が変わります。サンプルでは、ソースコ

Fig. 2-43　宝石のセルを操作する

ードが短くなるため、セルを入れ替える方法を使っています。

プログラム

List 2-8は宝石の順番を変えるプログラムです。宝石の移動処理を掲載しました。

宝石の順番を変える処理は、落下や左右への移動と同じく、入力状態で行います。ボタンの入力を検出したら、宝石のセルを操作して、順番を変えます。

List 2-8 宝石の順番を変える(CDroppingJewelJewelクラス)

```
// 宝石の移動処理
bool CDroppingJewelJewel::Move(const CInputState* is) {

    // 入力状態
    if (State==0) {
        // ...(中略)...

        // ボタンを入力しており、
        // かつ直前にボタンを離していたら、宝石の順番を変える
        if (is->Button[0] && !PrevButton) {

            // 一番上と中央の宝石を入れ替える
            CurrentJewel->Swap(0, 0, 0, 1);

            // 一番上と一番下の宝石を入れ替える
            CurrentJewel->Swap(0, 0, 0, 2);
        }

        // 直前のボタンの入力状態を保存する
        PrevButton=is->Button[0];

        // レバーを下に入れ続けたときに、宝石が次々に落ちないようにするための処理
        if (!is->Down) PrevDown=false;
    }
    // ...(中略)...
}
```

縦横斜めに揃える

同じ種類の宝石を縦横斜めに規定数以上揃えると、揃えた宝石が消えるというアクションです。移動や順番の変更を利用して、宝石を上手に積み上げ、消していくことがゲームの目的です。

Fig. 2-44　縦横斜めに揃える

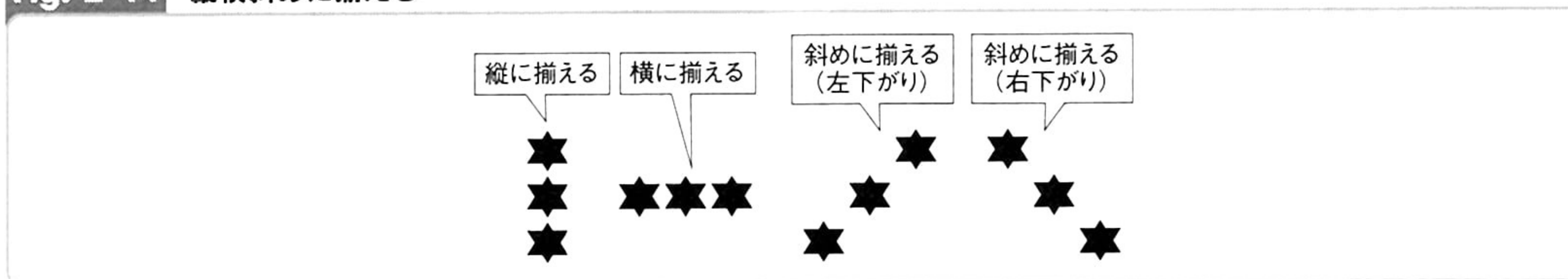
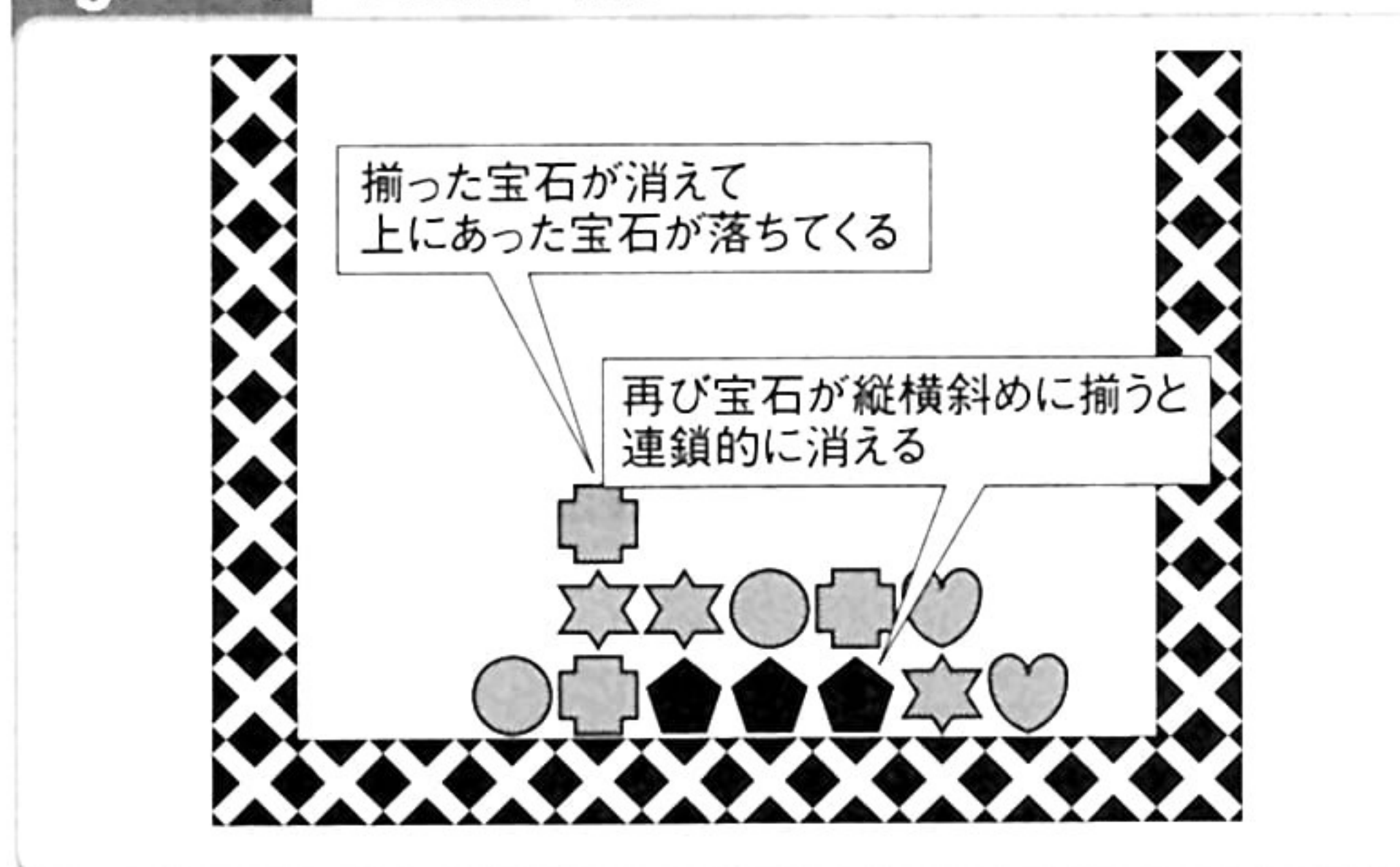



Fig. 2-45　縦横斜めのいずれかを同時に揃える

Fig. 2-46　連鎖的に消す

宝石が着地したときに、同じ種類の宝石が規定数以上揃っていると、揃った宝石が消えます。ここでは3個以上の宝石が揃ったときに消えることにしましょう。揃う方向は、縦横斜めのいずれかです（Fig. 2-44）。斜めには左下がりと右下がりの2種類があります。

縦横斜めのいずれかを同時に揃えてもかまいません。例えば、ある宝石が縦に揃っていると同時に、横や斜めにも揃っているような状態です（Fig. 2-45）。

揃った宝石が消えると、上にあった宝石が落ちてきます。このとき、再び宝石が縦横斜めに揃うと、連鎖的に消えます（Fig. 2-46）。

『コラムス』では、同じ種類の宝石を縦横斜めのいずれかに3個揃えると、消すことができます。多くの宝石を積み上げておき、一部の宝石を揃えて消すと、意外な場所で連鎖的に消えることがあります。これは『テトリス』とは違った楽しみです。

アルゴリズム

宝石を消すには、宝石が着地したときに、ステージのセルを調べます。消えるかどうかの条件は違いますが、基本的には「ブロックを1段揃えて消す」（→p. 64）と同じ方法です。

宝石が消えるのは、同じ種類の宝石が縦横斜めに規定数以上揃ったときです。ここでは3個以上揃ったときに消えることにします。4個や5個以上揃ったときに消すルールも、まったく同様の方法で実現できます。

ステージのすべてのセルについて順番に処理します。それぞれのセルから、下・右・左下・右下の4方向それぞれについて、同じ種類の宝石がいくつ並んでいるのかを数えます（Fig. 2-47）。

Fig. 2-47 同じ種類の宝石が並んでいる数を数える

=											=
=											=
=											=
											=
					1						
=					4						
=				3	2	0	3				=
=				1	2	2	4	2			=
=			0	3	2	4	2	1	2		=
=	=	=	=	=	=	=	=	=	=	=	=

このセルからは
同じ宝石が
下に3個、右下に3個
並んでいる

このセルからは
同じ宝石が
左下に2個、右下に2個
並んでいる

Fig. 2-48 消える宝石をマークする

=											=
=											=
=											=
=											=
=											=
=					1						=
=					4						=
=				3	2	0	3				=
=				1	2	2	4	2			=
=			0	3	2	4	2	1	2		=
=	=	=	=	=	=	=	=	=	=	=	=

同じ種類の宝石が
3個以上並んでいたら
消える宝石としてマークする

Fig. 2-49 消えていく宝石の表示

普通の宝石

消える宝石は
時間とともに
だんだん薄くする

同じ種類の宝石が3個以上並んでいたら、宝石を消すことができます。そこで、それらの宝石を消える宝石としてマークします（Fig. 2-48）。

マークすることによって、これから消える宝石と、普通の宝石を区別することができます。消える宝石を描画するときには、例えば時間とともにだんだん宝石の表示が薄くなって消えていくような演出をすると、どの宝石が消えたのかがわかりやすくなります（Fig. 2-49）。

連鎖的に宝石を消す

宝石が消えたら、上にあった宝石を下に落とします（Fig. 2-50）。そして、再びステージのすべてのセルについて、同じ種類の宝石が規定数以上並んでいるかどうかを調べます。

宝石が規定数以上並んでいるかぎり、宝石を消し、上にあった宝石を落とす処理を繰り返します。宝石が消えなくなったら、新しい宝石をステージ上方に出現させます。

宝石に関する処理の状態遷移をまとめたものがFig. 2-51です。宝石を消して上にあった宝石を落とす状態（消去状態）から、消える宝石を探す状態（着地状態）に戻ることがポイントです。これで、宝石が揃っているかぎり、連鎖的に消えるようになります。連鎖消しをしない場合は、消去状態から着地状態に戻らずに、再出現状態に移行します。

Fig. 2-50 上にあった宝石を落とす

=											=
=											=
=											=
=											=
=											=
=											=
=											=
=				3							=
=				1	1	0	3	2			=
=			0	3	4	4	4	1	2		=
=	=	=	=	=	=	=	=	=	=	=	=

消えた宝石の
上にあった宝石を落とす

同じ種類の宝石が
規定数以上並んで
いるかどうかを
再び調べる

Fig. 2-51 宝石の処理に関する状態遷移

入力状態
レバー入力で宝石を左右に動かす
ボタン入力で宝石の順番を変える
時間で宝石を落下させる

宝石が床や他の宝石に着地した

左右に移動した、または落下した

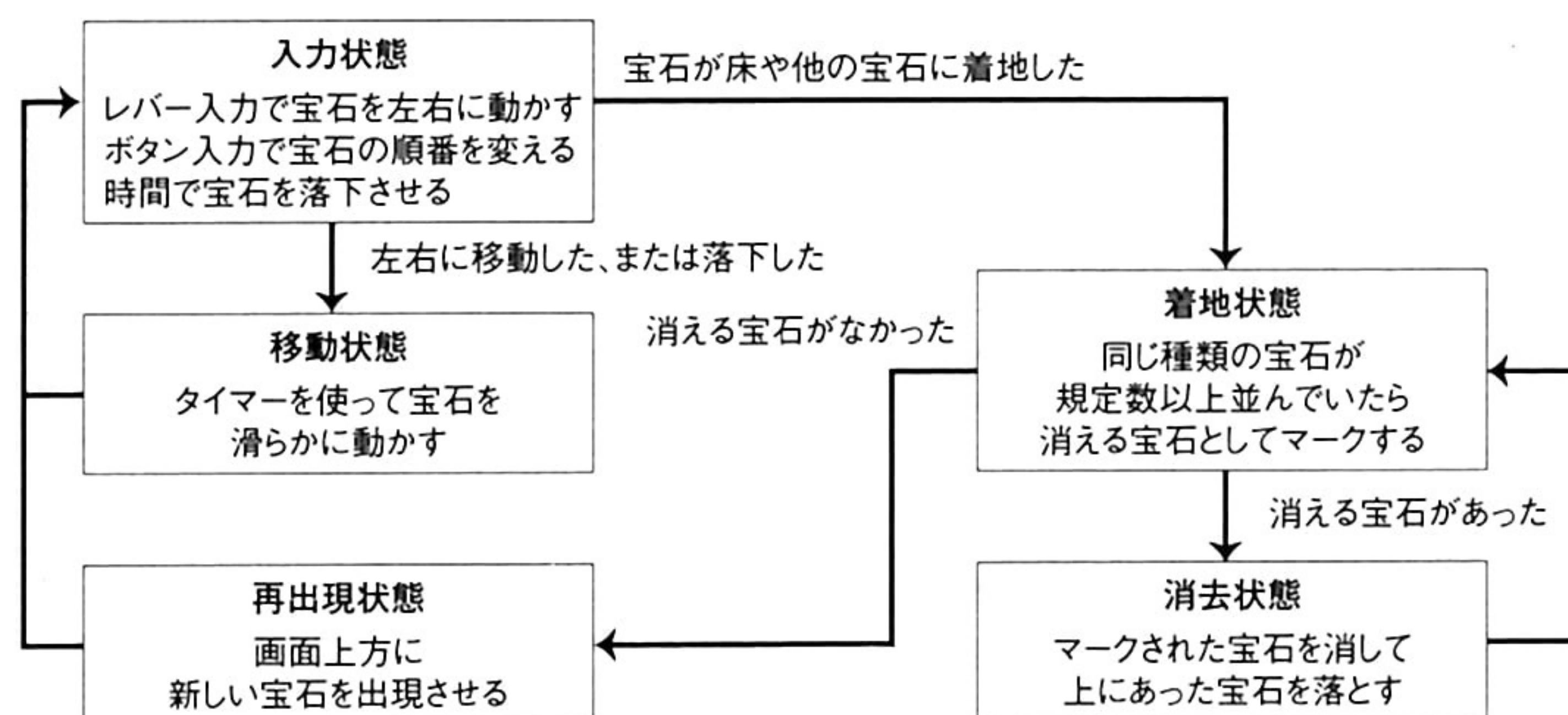
移動状態
タイマーを使って宝石を
滑らかに動かす

消える宝石がなかった

着地状態
同じ種類の宝石が
規定数以上並んでいたら
消える宝石としてマークする

消える宝石があった

再出現状態
画面上方に
新しい宝石を出現させる

消去状態
マークされた宝石を消して
上にあった宝石を落とす

プログラム

List 2-9は宝石を縦横斜めに揃えて消すプログラムです。宝石の移動処理と描画処理を掲載しました。

移動処理は、着地状態・消去状態・再出現状態に分かれています。着地状態では、同じ種類の宝石がいくつ並んでいるのかを調べ、規定数以上並んでいたら、消える宝石としてマークします。消去状態では、マークされた宝石を消し、上にあった宝石を下に落とします。再出現状態では、新しい宝石をステージ上方に出現させます。

消える宝石のマークは、セルの最上位ビットをセットする（ビットを1にする）ことで行います。ここでは最上位ビットをセットしたりクリアしたりといった操作が簡単だったので採用しましたが、他の方法でマークを付けてもかまいません。

並んでいる宝石を数えるときには、「下・右・左下・右下」の4方向について調べます。縦横斜めというと、上・左・右上・左上も含めた8方向を調べる必要がありそうですが、4方向で十分です。ステージのすべてのセルについて調べるので、例えばあるセルから下方向へ調べれば、

そのセルよりも下にあるセルから上方向へ調べたのと同じことになるからです。

描画処理では、消える宝石・普通の宝石・ステージの壁を描画します。消える宝石は、タイマーを使って、時間とともに少しずつ色が薄くなるように表示します。

List 2-9 縦横斜めに揃える(CDroppingJewelJewelクラス)

```
// 宝石の移動処理
bool CDroppingJewelJewel::Move(const CInputState* is) {

    // ...(中略)...

    // 着地状態
    if (State==2) {

        // 宝石がまったく消えなかった場合には、
        // 再出現状態に移行する
        State=4;

        // ステージ内のすべてのセルについて調べる
        for (
            int y=DROPPING_JEWEL_FIELD_TOP;
            y<DROPPING_JEWEL_FIELD_BOTTOM;
            y++
        ) {
            for (
                int x=DROPPING_JEWEL_FIELD_LEFT;
                x<DROPPING_JEWEL_FIELD_RIGHT;
                x++
            ) {
                // セルを取得する
                // セルの種類を比較するために、
                // 消える宝石のマーク(最上位ビット)をクリアする
                char c=StageCell->Get(x, y)&0x7f;

                // セルが空の場合には次のセルに進む
                if (c==' ') continue;

                // 宝石が並ぶ方向
                static const int
                    vx[]={0, 1, -1, 1},
                    vy[]={1, 0, 1, 1};

                // 縦横斜め(4方向)について、
                // 宝石が並んでいるかどうかを調べる
                for (int v=0; v<4; v++) {

                    // 並んでいる宝石の個数
                    int count=0;
```

```
					//1つの方向について、
					//同じ宝石のセルがいくつ並んでいるのかを調べる
					for (
						int i=x, j=y;
						DROPPING_JEWEL_FIELD_LEFT<=i &&
						i<DROPPING_JEWEL_FIELD_RIGHT &&
						DROPPING_JEWEL_FIELD_TOP<=j &&
						j<DROPPING_JEWEL_FIELD_BOTTOM;
						i+=vx[v], j+=vy[v], count++
					) {
						//同じ宝石が並んでいるかぎりループする
						if (c!=(StageCell->Get(i, j)&0x7f)) break;
					}

					//並んでいる宝石の数が規定値(3個)以上ならば、
					//消える宝石のマーク(最上位ビット)をセットする
					if (count>=DROPPING_JEWEL_ERASING_COUNT) {
						for (
							int i=x, j=y, k=0;
							k<count;
							i+=vx[v], j+=vy[v], k++
						) {
							StageCell->Set(
								i, j, StageCell->Get(i, j)|0x80);
						}

						//タイマーの設定
						Time=0;

						//消去状態に移行する
						State=3;
					}
				}
			}
		}
	}

	//消去状態
	if (State==3) {

		//タイマーの更新
		Time++;

		//タイマーが一定値に達したら、
		//マークされた宝石を消去する
		if (Time==20) {

			//ステージ内のすべてのセルについて処理する
			for (
```

```
                int x=DROPPING_JEWEL_FIELD_LEFT;
                x<DROPPING_JEWEL_FIELD_RIGHT;
                x++
            ) {
                for (
                    int y=DROPPING_JEWEL_FIELD_TOP;
                    y<DROPPING_JEWEL_FIELD_BOTTOM;
                    y++
                ) {
                    //消える宝石を見つけたら消去し、
                    //その宝石よりも上にあるすべての宝石を、
                    //1段ずつ下に落とす
                    if (StageCell->Get(x, y)&0x80) {
                        for (
                            int i=y;
                            i>=DROPPING_JEWEL_FIELD_TOP;
                            i--
                        ) {
                            StageCell->Set(
                                x, i, StageCell->Get(x, i-1));
                        }
                    }
                }
            }

            //着地状態に移行する
            //宝石が消えたことによって、
            //新たに宝石が並ぶ可能性があるため
            State=2;
        }
    }

    //再出現状態
    if (State==4) {

        //新しい宝石をステージ上方に出現させる
        Init();
    }

    return true;
}

//宝石の描画処理
void CDroppingJewelJewel::Draw() {

    //画面の解像度を取得する
    float
        sw=Game->GetGraphics()->GetWidth()/MAX_X,
        sh=Game->GetGraphics()->GetHeight()/MAX_Y;
```

```
    // …（中略）…

    // ステージのすべてのセルについて処理する
    for (int y=0; y<MAX_Y; y++) {
        for (int x=0; x<MAX_X; x++) {

            // セルの種類を取得する
            char c=StageCell->Get(x, y);

            // 消える宝石
            if (c&0x80) {

                // タイマーの値に応じて色を決める
                float f=Time*0.05f;

                // 時間とともに少しずつ色が薄くなるように描画する
                Game->Texture[TEX_BALL0+((c&0x7f)-'0')]->Draw(
                    x*sw, y*sh, sw, sh,
                    0, 0, 1, 1, D3DXCOLOR(f, f, f, 1)
                );
            } else

            // 普通の宝石
            if ('0'<=c && c<'0'+DROPPING_JEWEL_COLOR_COUNT) {

                // 宝石の種類に応じたグラフィックを描画する
                Game->Texture[TEX_BALL0+(c-'0')]->Draw(
                    x*sw, y*sh, sw, sh, 0, 0, 1, 1, COL_BLACK
                );
            } else

            // 壁
            if (c=='=') {
                Game->Texture[TEX_DROP_FLOOR]->Draw(
                    x*sw, y*sh, sw, sh, 0, 0, 1, 1, COL_BLACK
                );
            }
        }
    }
}
```

ボールを落とす

ボールを落として積み上げるアクションです。レバー入力でボールを左右に移動し、ボタン入力でボールを回転させることができます。同じ種類のボールを規定数以上隣接させて積むと、ボールを消すことができます。

ここではボールが2個1組で落ちてくる場合を考えます（Fig. 2-52）。ボールの種類はランダムに決まります。ボールは異なる種類の組み合わせで落ちてくることも、同じ種類の組み合わせで落ちてくることもあります。

ボールは時間とともに少しずつ落下します。レバーを左右に入力すると、ボールを左右に動かすことができます。また、レバーを下に入力すると、ボールを速く落下させることができます。

ステージの床に落ちるか、積み上げられた他のボールの上に落ちると着地します（Fig. 2-53）。着地したときに、同じ種類が規定数以上隣接していると、ボールは消えます。

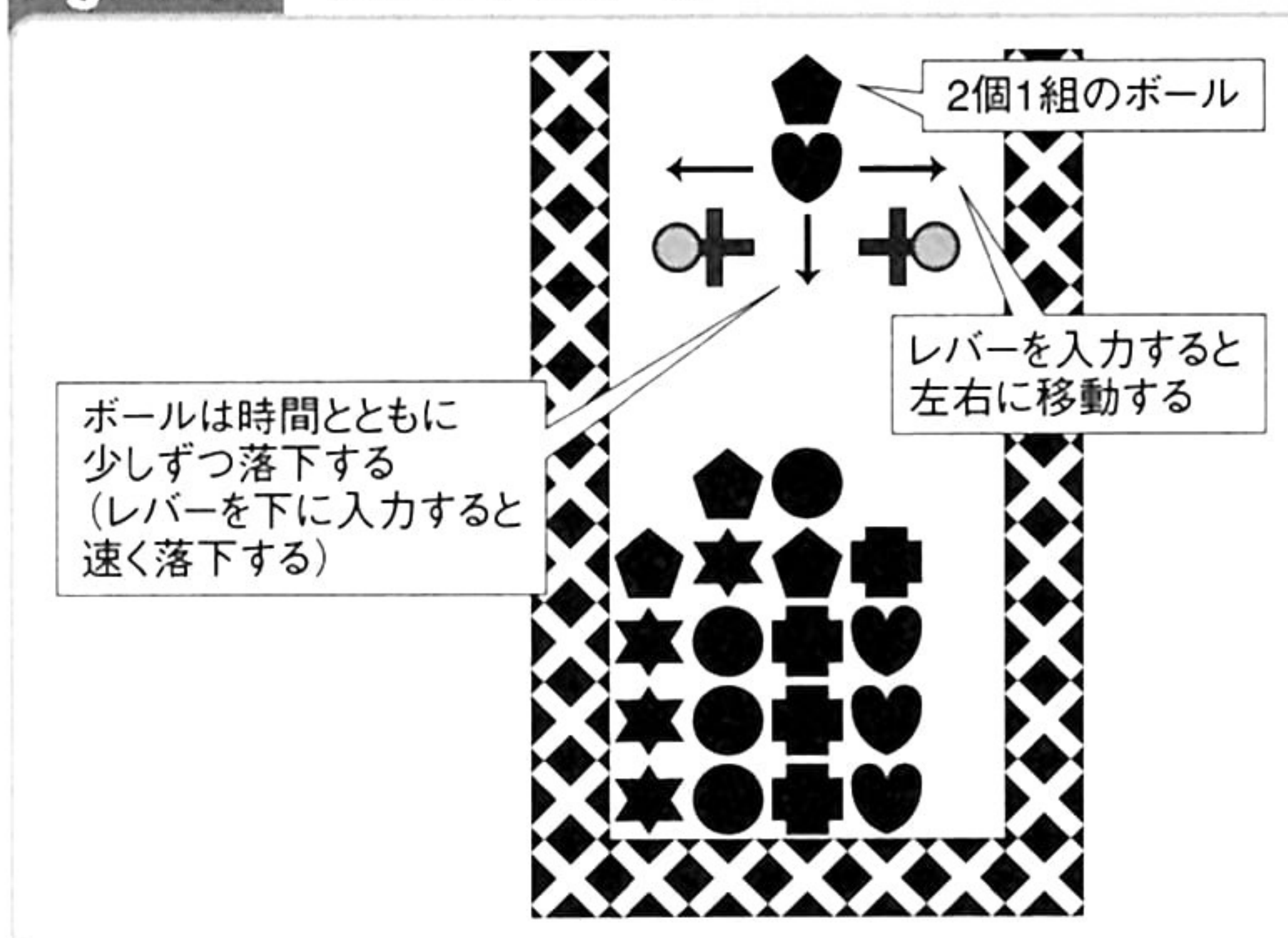
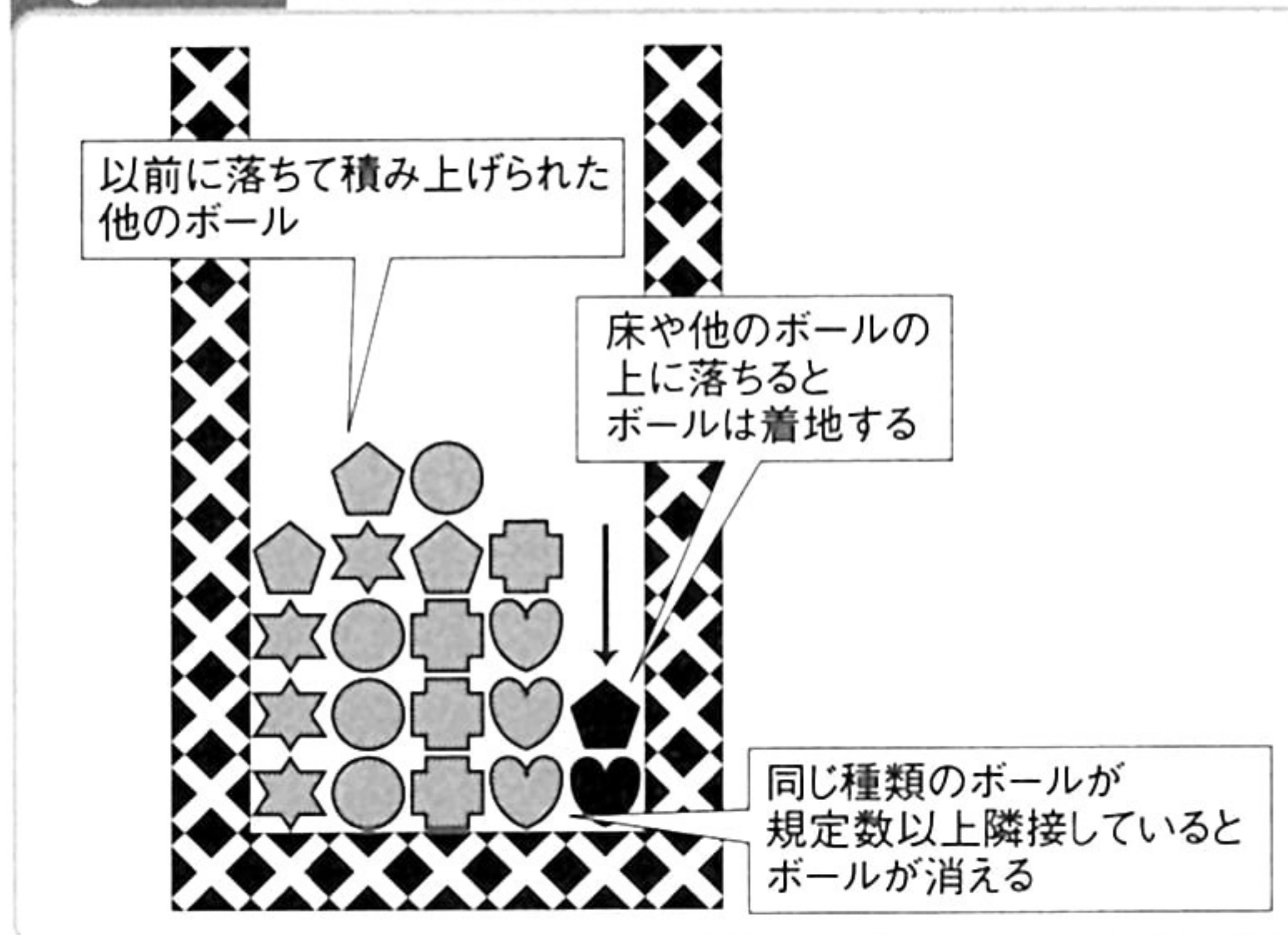
『ぷよぷよ』はボール（正確にはボール状の物体ですが）を落として積み上げるゲームです。ブロックを落とす『テトリス』や、宝石を落とす『コラムス』の流れをくむゲームですが、ボールを上手に積み上げることによって、大きな連鎖を組むことができるのが特徴です。ボールを1つ1つ消していくことよりも、大きな連鎖を組む方が有利なので、爽快感のあるゲームに仕上がっています。なお、『ぷよぷよ』ではボールの形はすべて同じで、色が異なります。同じ色のボールを規定数（4個）以上隣接させると、消すことができます。本書では白黒でもわかりやすいように、ボールの色ではなく形状を変えて示すことにしました。

ボールを落とすゲームは『ぷよぷよ』以外にも非常に多くあります。『対戦ぱずるだま』は『ぷよぷよ』に比較的よく似たゲームです。『戦球』もよく似ていますが、ボールを互い違いに積み上げる点がユニークです。少し変わっているのは『もうぢゃ』で、ボールのかわりに1円や10円といったコインを落とします。コインを上位のコインに両替することによって、消すことができます。

Fig. 2-52 落ちてくるボール

2個1組のボール
レバーを入力すると左右に移動する
ボールは時間とともに少しずつ落下する（レバーを下に入力すると速く落下する）

Fig. 2-53 ボールの着地

以前に落ちて積み上げられた他のボール
床や他のボールの上に落ちるとボールは着地する
同じ種類のボールが規定数以上隣接しているとボールが消える

アルゴリズム

ボールを落とすアクションは、「ブロックを落とす」(→p. 50) などと同じ方法で実現できます。左右への移動についても、「ブロックを左右に移動する」(→p. 57) と同様です。

落ちてくるボールはセルで表現します (Fig. 2-54)。ここでは「3×3」のセルを使いましょう。ボールは種類に応じて「0～4」の数字で表します。

ボールは2個だけなのに、3×3のセルを使うのは、ボールを回転させるためです。詳しくは「障害物を避けながらボールを回転させる」(→p. 95) で解説します。

ステージもセルで表現します (Fig. 2-55)。ステージの周囲を囲む壁は文字「＝」で表し、積み上げたボールは「0～4」の数字で表します。

ボールを時間とともに落下させるには、タイマーを使います (Fig. 2-56)。タイマーを更新し、タイマーが一定値になるたびに、ボールのセル座標を更新します。レバーを下に入力したときには、タイマーが一定値になるのを待たずに、セル座標を更新します。

Fig. 2-54 ボールをセルで表現する

ボールを数字 (0～4) の
セルで表現する

落ちてくるボールを
3×3のセルで表現する

Fig. 2-55 ステージをセルで表現する

=						=
=						=
=						=
=						=
=						=
=		4	0			=
=	4	1	4	3		=
=	1	0	3	2		=
=	1	0	3	2		=
=	1	0	3	2		=
=	=	=	=	=	=	=

周囲を囲む壁
(＝)

積み上げられたボール
(0～4)

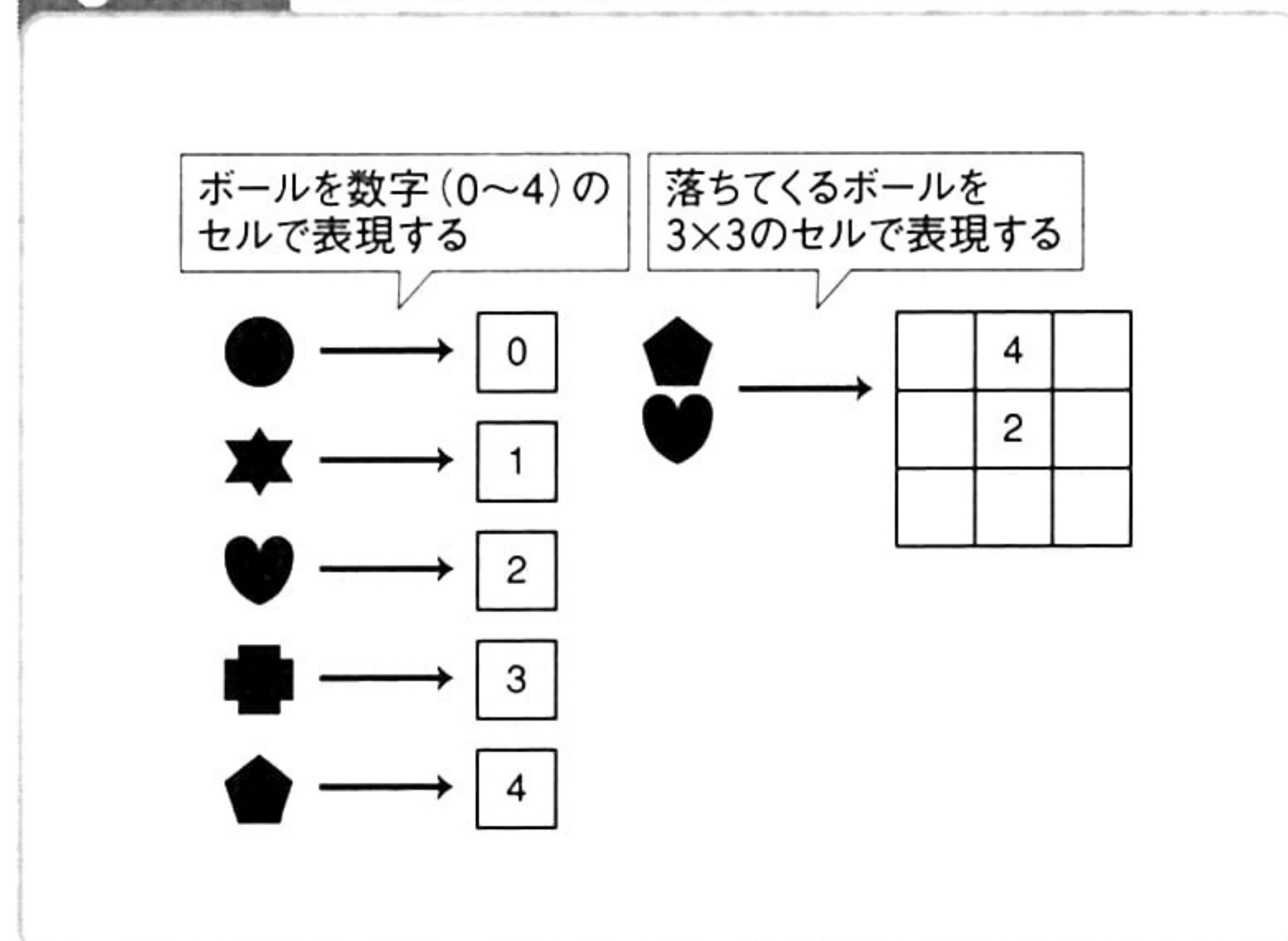
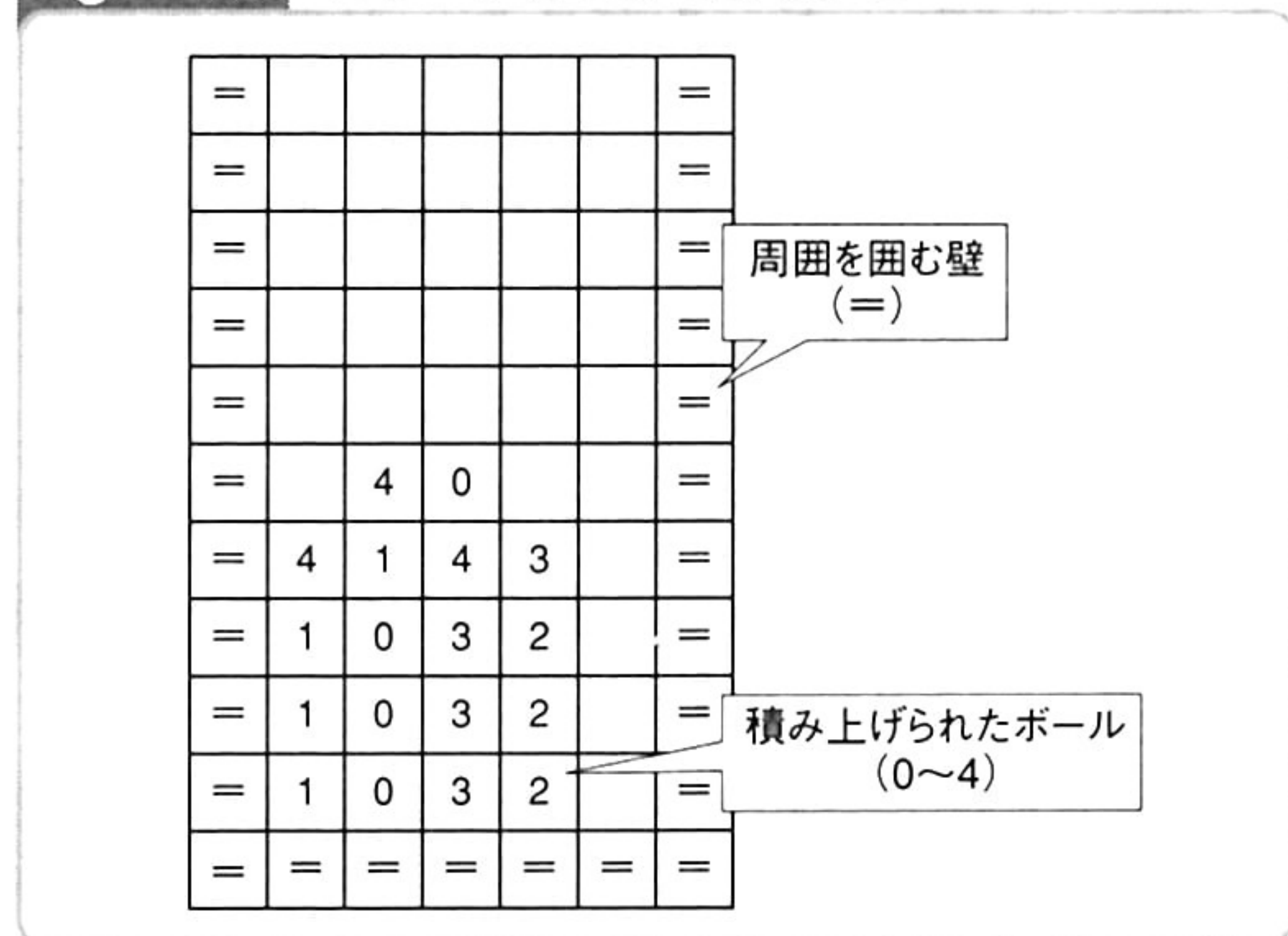



Fig. 2-56 時間とともにボールを落下させる

セル座標
(CX, CY)

一定時間が経過する
(タイマーを使用)

セル座標を更新
(CX, CY+1)

レバーを下に入力したときには
タイマーを待たずに
すぐにセル座標を更新する

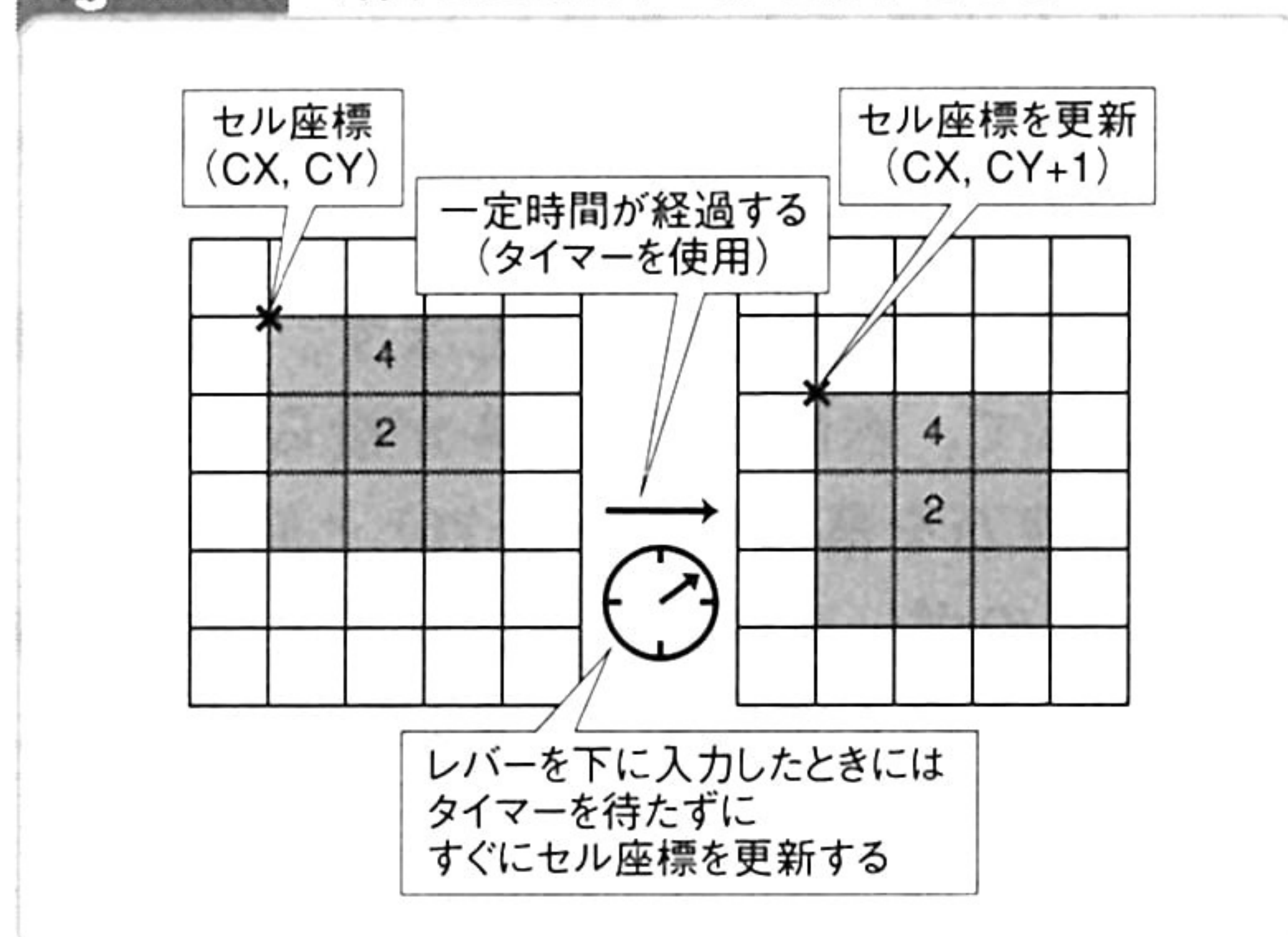



Fig. 2-57 ボールとステージの当たり判定処理

ボールのセル
(3×3)

ボールの空ではないセルと
ステージの空ではないセルとの
接触を調べる

接触しないかぎりは
ボールを落下させる
ことができる

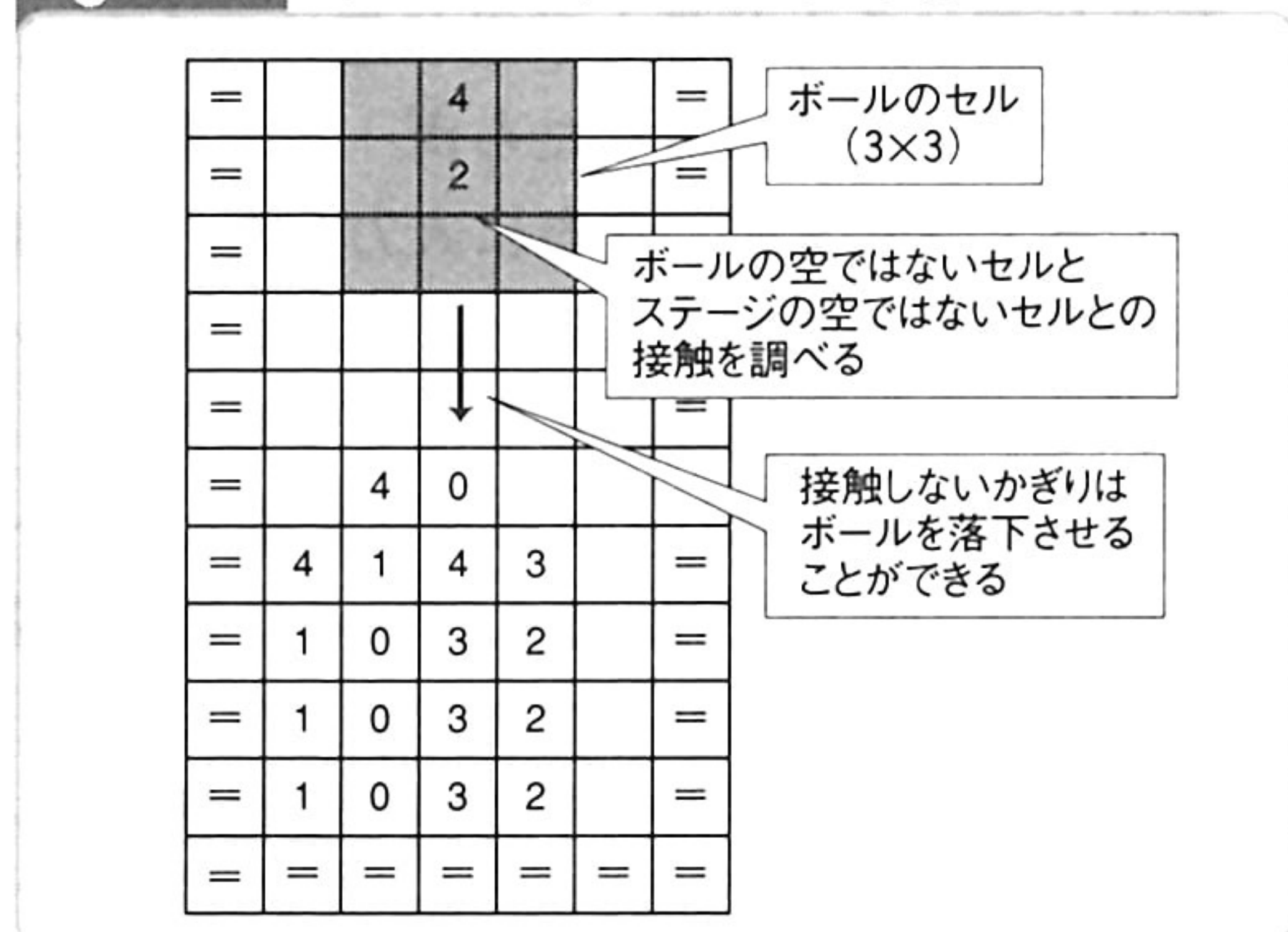



Fig. 2-58 ボールの着地

=						=
=						=
=						=
=						=
=						=
=		4	0			=
=	4	1	4	3		=
=	1	0	3	2		=
=	1	0	3	2	4	=
=	1	0	3	2	2	=
=	=	=	=	=	=	=

ボールが床や他のボールに接触して
落下させることができなくなったら
着地させる

Fig. 2-59 左右への移動

=			4			=
=	←		2	→		=
=						=
=						=
=						=
=		4	0			=
=	4	1	4	3		=
=	1	0	3	2		=
=	1	0	3	2		=
=	1	0	3	2		=
=	=	=	=	=	=	=

接触しないかぎりは
ボールを左右に移動できる

ボールの空ではないセルと
ステージの空ではないセルとの
接触を調べる

ボールを落とす際には、ボールのセルとステージのセルとの間で、当たり判定処理を行います（Fig. 2-57）。ボールがステージの床や他のボールに接触しない場合にかぎって、ボールを落とします。

ボールが床や他のボールに接触する場合には、ボールを着地させます（Fig. 2-58）。着地したボールのセルは、ステージのセルに合成します。

レバーを左右に入力したときには、ボールを左右に移動します。移動の前に、ボールのセルとステージのセルとの間で当たり判定処理を行い、接触しないときだけ、セル座標を更新します（Fig. 2-59）。接触する場合には、ボールを移動させません。

プログラム

List 2-10はボールを落とすプログラムです。ボールの移動処理を掲載しました。

移動処理は入力状態と移動状態に分かれています。入力状態では、落下タイマーを更新し、タイマーが一定値に達したらボールを落下させます。レバーを下に入力したときには、ボールを速く落下させます。左右に入力したときには、ボールを左右に移動させます。

ボールを落下させるときと、左右に移動させるときには、移動状態に移行します。移動状態では、ボールの描画座標を少しずつ変化させることによって、画面上でボールが滑らかに動くようにします。これは「ブロックを落とす」（→p. 50）や「宝石を落とす」（→p. 76）と同様です。

移動処理の冒頭では、2人プレイに対応するために、このボールが属するプレイヤーの入力を取得しています。サンプルでは、プレイヤーごとにボールのオブジェクトを1個ずつ作成し、各プレイヤーのフィールドに関する処理を行っています。

List 2-10　ボールを落とす(CDroppingBallBallクラス)

```
// ボールの移動処理
bool CDroppingBallBall::Move(const CInputState*) {

    // 2人プレイに対応するために、
    // このボールが属するプレイヤーの入力を取得する
    const CInputState* is=Game->GetInput()->GetState(PlayerID);

    // 入力状態
    if (State==0) {

        // 落下タイマーの更新
        DropTime++;

        // レバーを下に入力するか、
        // 落下タイマーが一定値に達したら、
        // ボールを落下させる
        if ((is->Down && !PrevDown)|| DropTime==120) {

            // ステージに接触したら、ボールを着地させる
            if (StageCell->Hit(CX, CY+1, CurrentBall)) {

                // ステージのセルにボールのセルを合成する
                StageCell->Merge(CX, CY, CurrentBall);

                // 連鎖カウント(連鎖消しの回数)を初期化する
                ChainCount=0;

                // 落下判定状態に移行する
                State=2;
            } else

            // ステージに接触しない場合には、
            // ボールを1段落下させる
            {
                // 落下タイマーの設定
                DropTime=0;

                // セル座標の更新
                CY++;

                // 速度・タイマーの設定
                VX=0;
                VY=1;
                Time=0;

                // 移動状態に移行する
                State=1;
            }
```

```
    } else

    // レバーを左に入力しており、
    // かつステージに接触しないならば、
    // 左に移動する
    if (is->Left && !StageCell->Hit(CX-1, CY, CurrentBall)) {

        // セル座標の更新
        CX--;

        // 速度・タイマーの設定
        VX=-1;
        VY=0;
        Time=0;

        // 移動状態に移行する
        State=1;
    } else

    // レバーを右に入力しており、
    // かつステージに接触しないならば、
    // 右に移動する
    if (is->Right && !StageCell->Hit(CX+1, CY, CurrentBall)) {

        // セル座標の更新
        CX++;

        // 速度・タイマーの設定
        VX=1;
        VY=0;
        Time=0;

        // 移動状態に移行する
        State=1;
    } else

    // …(中略)…
}

// 移動状態
if (State==1) {

    // タイマーの更新
    Time++;

    // 描画座標の更新
    X=CX-VX*(1-Time*0.1f);
    Y=CY-VY*(1-Time*0.1f);

    // タイマーが一定値に達したら、
```

```
            // 入力状態に移行する
            if (Time==10) {
                State=0;
            }
        }

        // …（中略）…
    }
```

SAMPLE

「DROPPING BALL」は「ボールを落とす」「障害物を避けながらボールを回転させる」「着地したボールが2つに分かれる」「連鎖的に消す」「相手側にボールを降らせる」のサンプルです。

レバーの左右（カーソルキーの左右、プレイヤー2はJキーとLキー）でボールが移動し、下（カーソルキーの下、プレイヤー2はKキー）でボールの落下スピードが上がります。ボタン0（Zキー、プレイヤー2は;キー）を押すと、ボールが右回りに回転します。

同じ種類のボールを4個以上隣接させると、ボールを消すことができます。ボールを消すと、上に載っていたボールが落下して、再び同じ種類のボールが隣接し、連鎖的に消えることがあります。

左側のフィールドはプレイヤー1、右側のフィールドはプレイヤー2が使います。画面中央に表示されているボールは、次に落ちてくるボールです。左側の2個がプレイヤー1のボール、右側の2個がプレイヤー2のボールです。

ボールを消した数や連鎖数に応じて、相手側のフィールドに透明の攻撃ボールが降ります。攻撃ボールは、隣接する通常のボールを消したときにかぎって、消すことができます。連鎖が大きいと、相手側に降る攻撃ボールの数が爆発的に増えます。上手にボールを積み上げることによって、素早く大連鎖を作ることが、このゲームを攻略するポイントです。

DROPPING BALL → p. 387

障害物を避けながらボールを回転させる

ボタン入力でボールを回転させるアクションです。回転させることによって、ボールを積み上げるときに、ボールの種類を揃えやすくなります。さらに、快適にプレイできるように、ボールを回転する際にステージの壁や他のボールを避ける機能についても解説します。

ボタンを入力すると、ボールが回転します（Fig. 2-60）。ボールは片方のボールを中心として、90度ずつ回転します。4回ボタンを押すと、元の配置に戻ります。ここまでは「ブロックを回転させる」（→p. 60）に似ています。

ここで、ボールをステージの壁近くで回転させたり、積み上げられた他のボールの近くで回転させたりする場合を考えましょう。例えば、ボールを回転させるときに、右にある壁や他のボールがジャマになったとします（Fig. 2-61）。障害物を避けるためには、まずボールを左に移動してから、ボールを回転させる必要があります。

これはプレイヤーにとってやや複雑で、直感的にわかりにくい操作です。そこで、この手順を自動的に行います(Fig. 2-62)。ボールが回転するときに、右に障害物があり、左が空いているときには、左に避けつつ回転します(左右とも空いていない場合は回転できません)。これでプレイヤーの操作はとても簡単になり、快適に遊べるようになります。

左に障害物がある場合も同様です。右が空いているときには、ボールは右に避けつつ回転します(Fig. 2-63)。

Fig. 2-60 ボールを回転させる

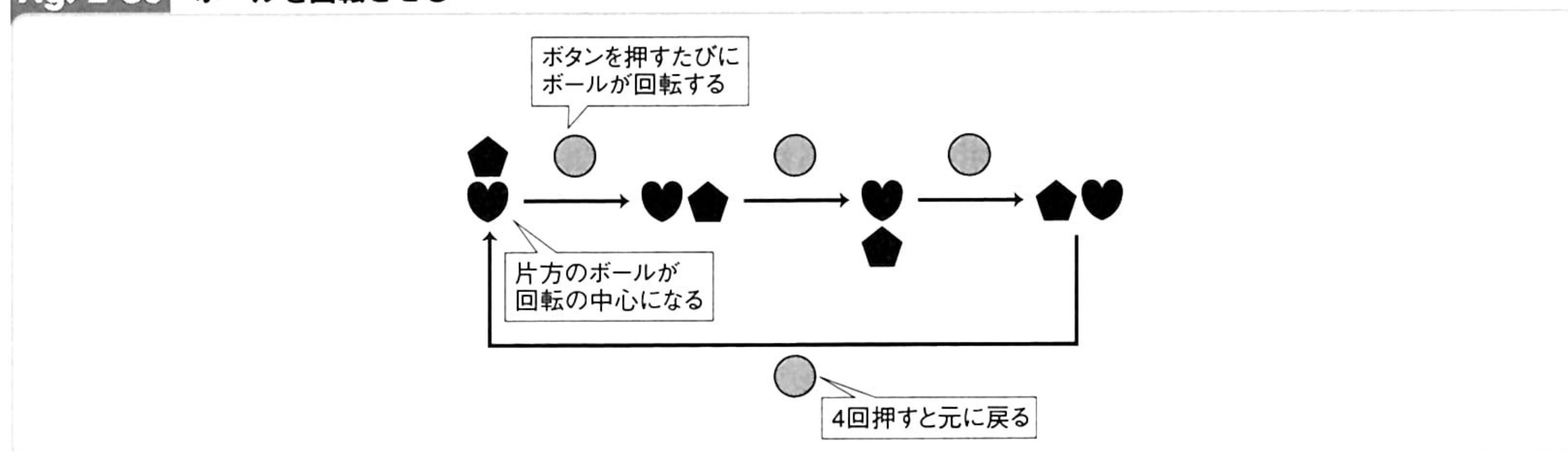



Fig. 2-61 ボールの右に障害物がある場合

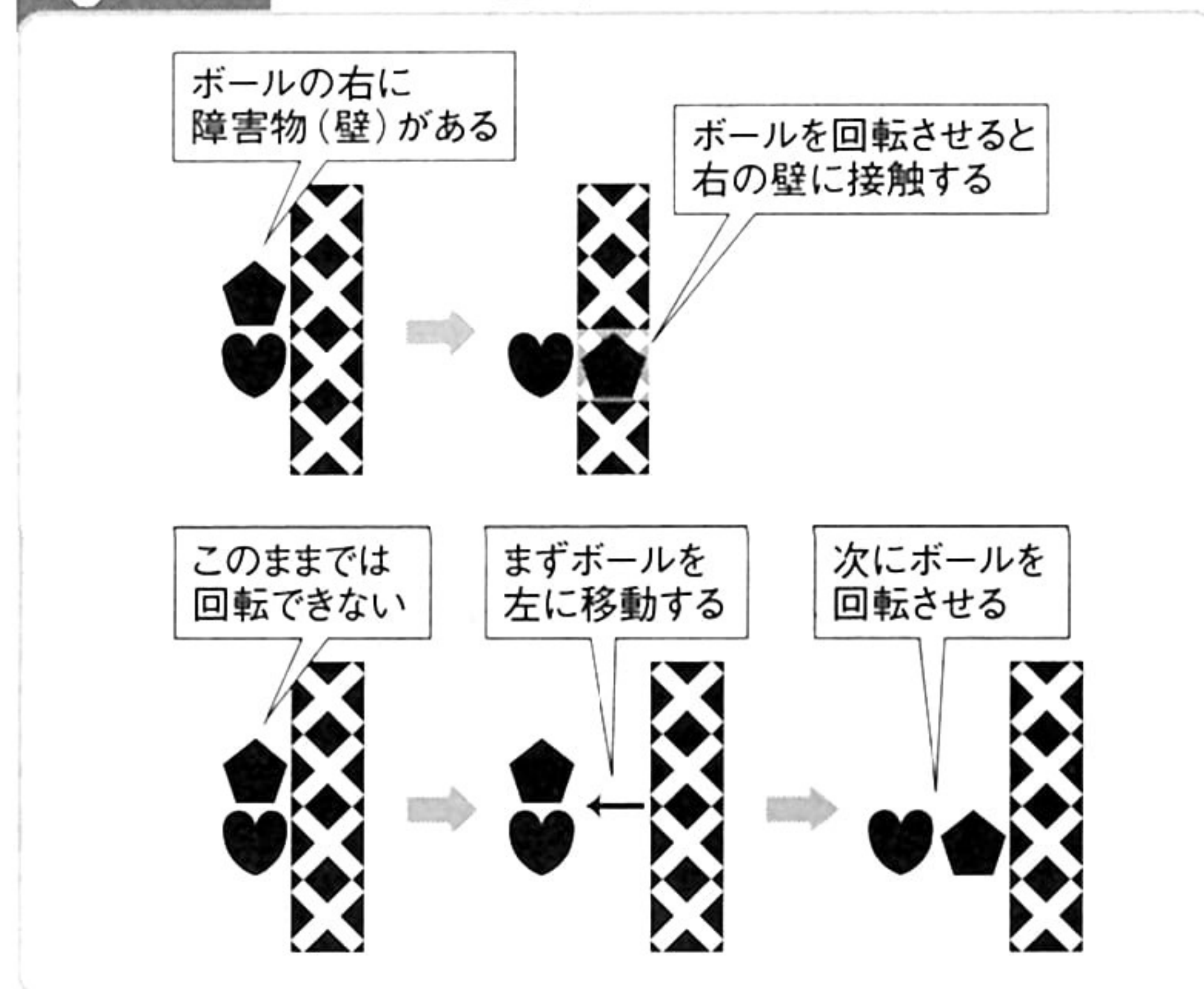



Fig. 2-62 ボールが自動的に障害物を避けて回転する

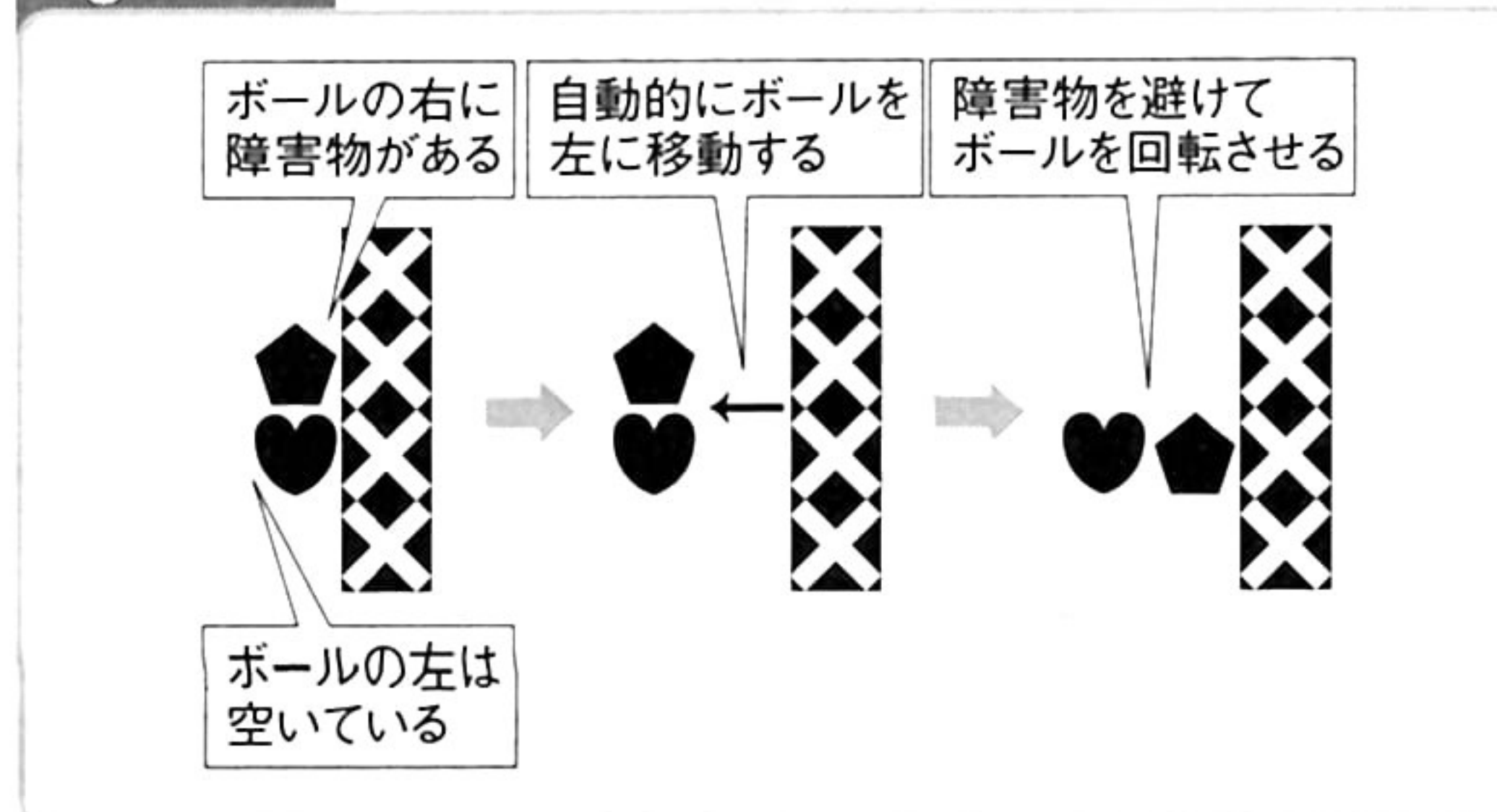



Fig. 2-63 ボールの左に障害物がある場合

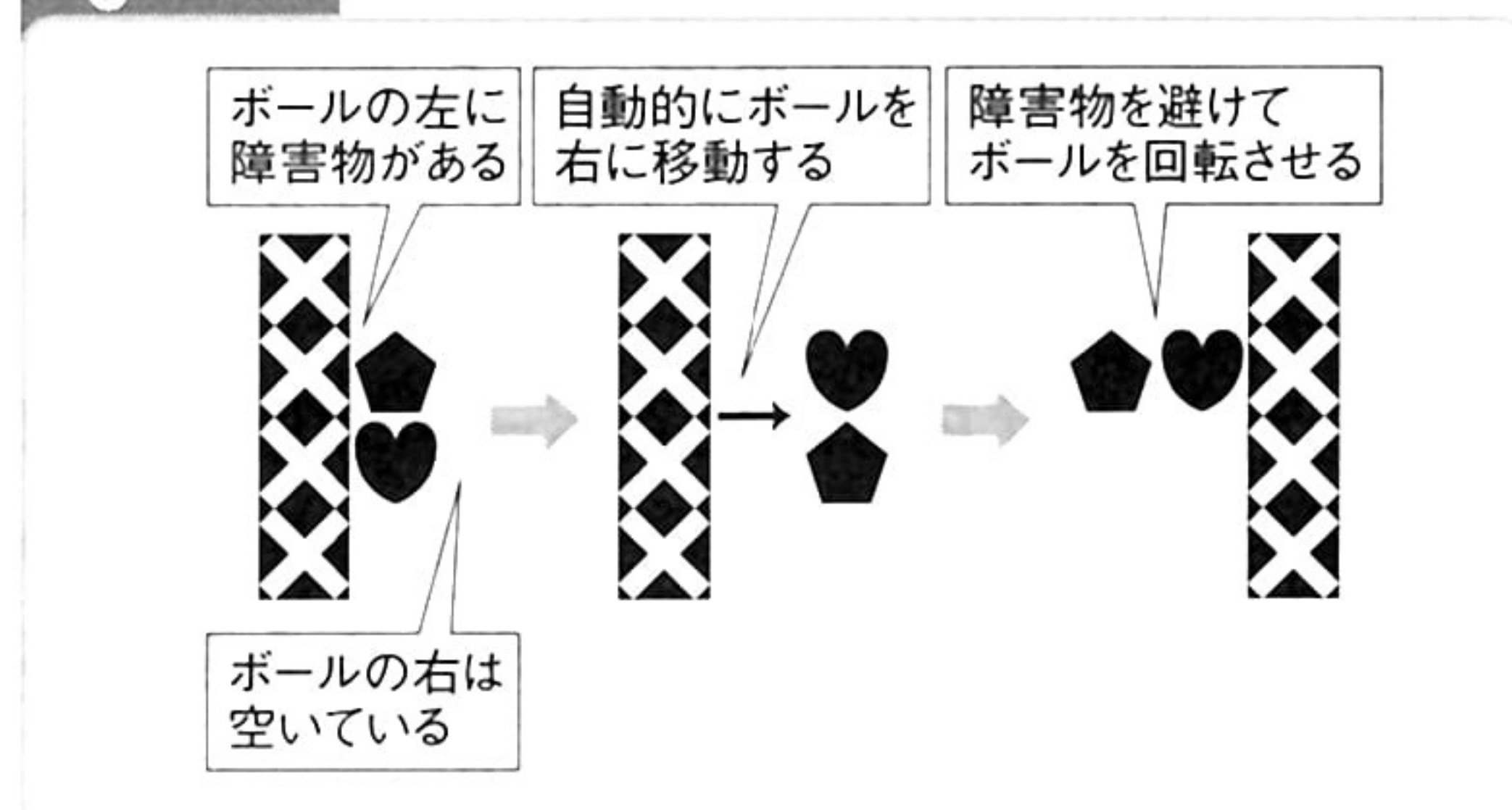
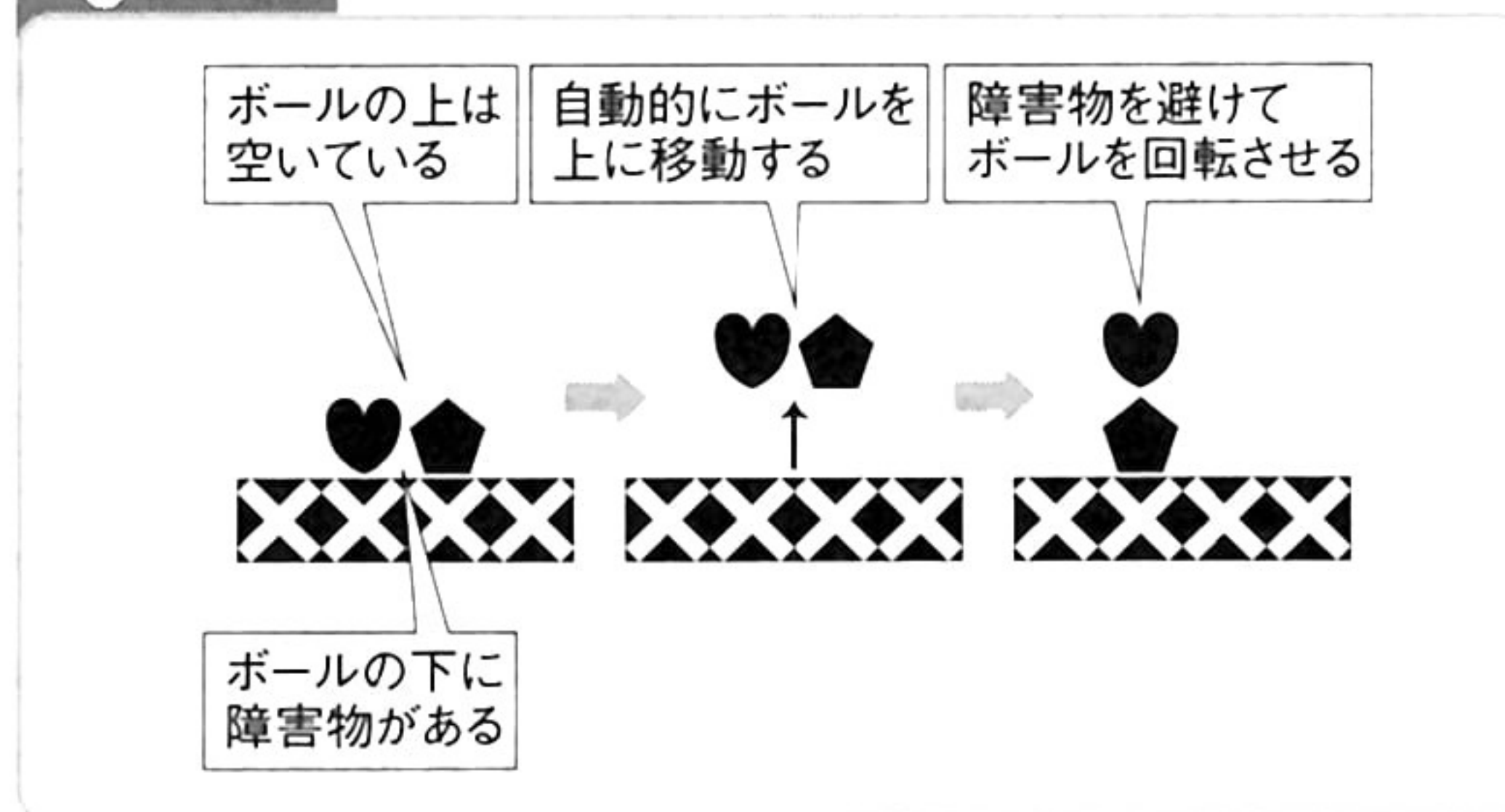



Fig. 2-64 ボールの下に障害物がある場合

下に障害物がある場合も、ボールは似たような動きをします。上が空いているならば、ボールは上に避けつつ回転します（Fig. 2-64）。

障害物を避けながらボールを回転させる機能は、『ぷよぷよ』などに採用されています。この機能があると、ボールの回転が非常に操作しやすくなり、より繊細なプレイが可能になります。

『テトリス』にはこの機能がないため、壁際でブロックを回転させづらいことがあります。続編の『テトリス ザ・グランドマスター』などには、ブロックが障害物を避けて回転する機能があり、プレイしやすくなっています。

アルゴリズム

ボールを回転させるには、ボールのセルを操作します（Fig. 2-65）。中央のセルはそのままにして、上下左右のセルを右回りに回転させます。

セルを入れ替えても同じ結果が得られます。上のセルと左のセルを入れ替え、左のセルと下のセルを入れ替え、最後に下のセルと右のセルを入れ替えます。これで、上下左右のセルを右回りに回転させた状態になります。

Fig. 2-65 ボールのセルを操作する

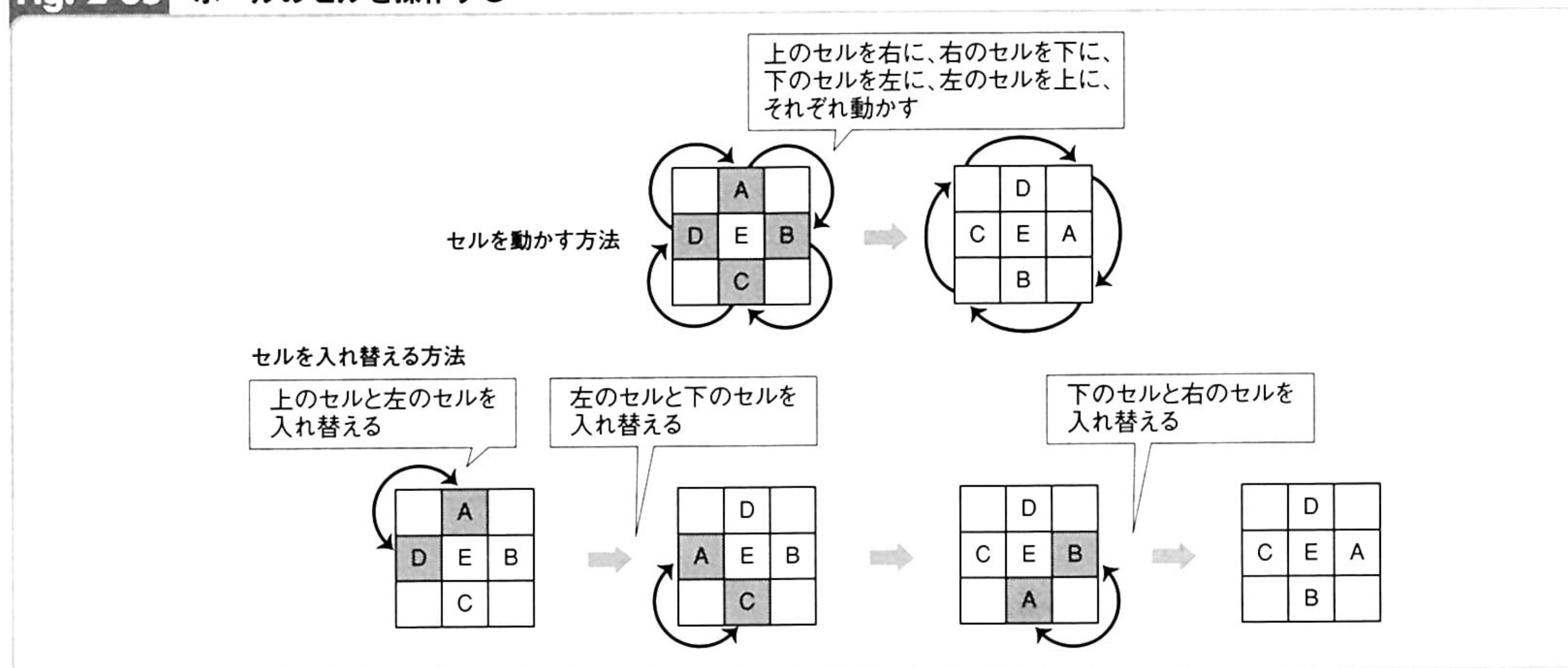



Fig. 2-66 右の障害物を避ける

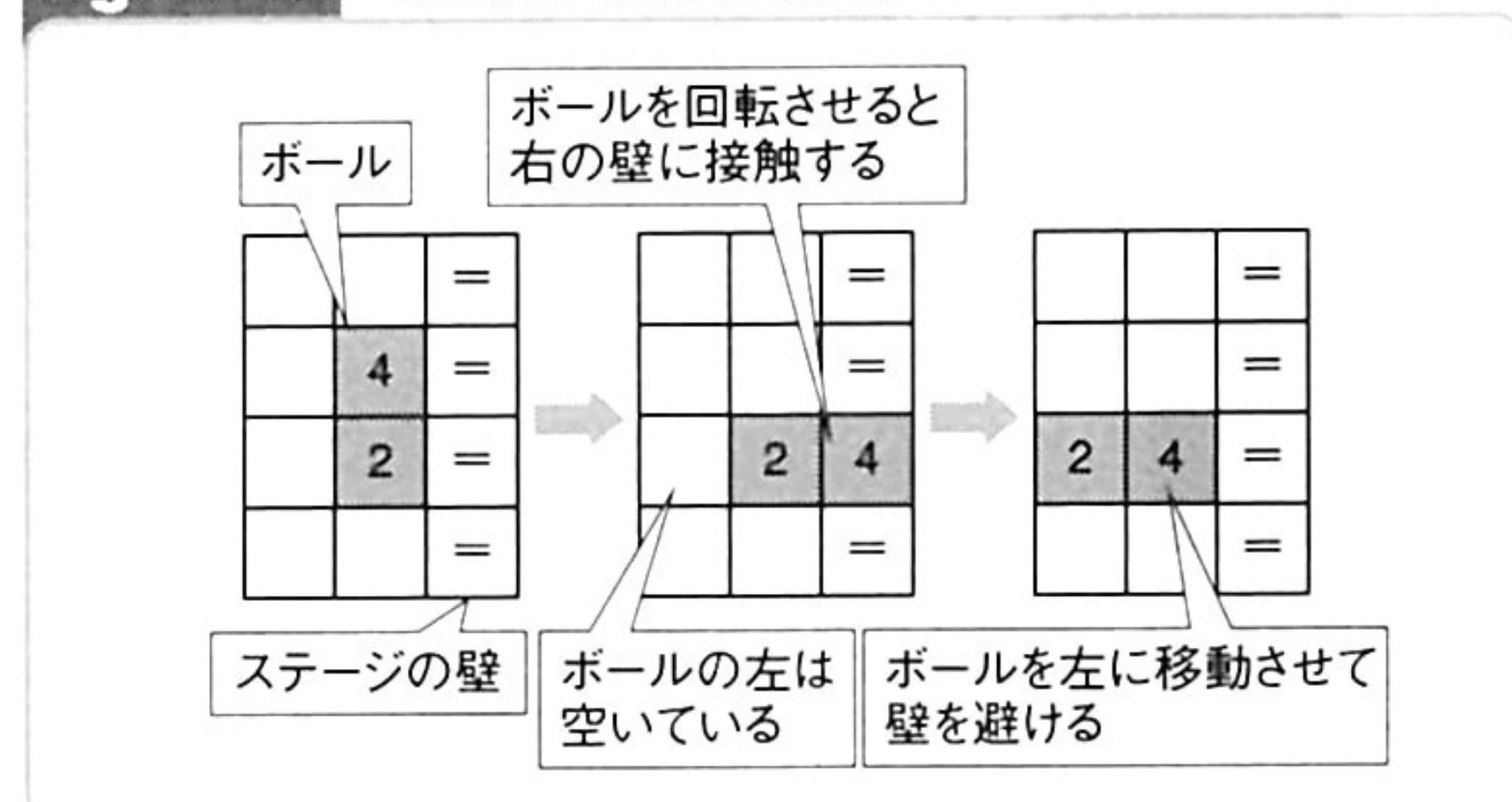
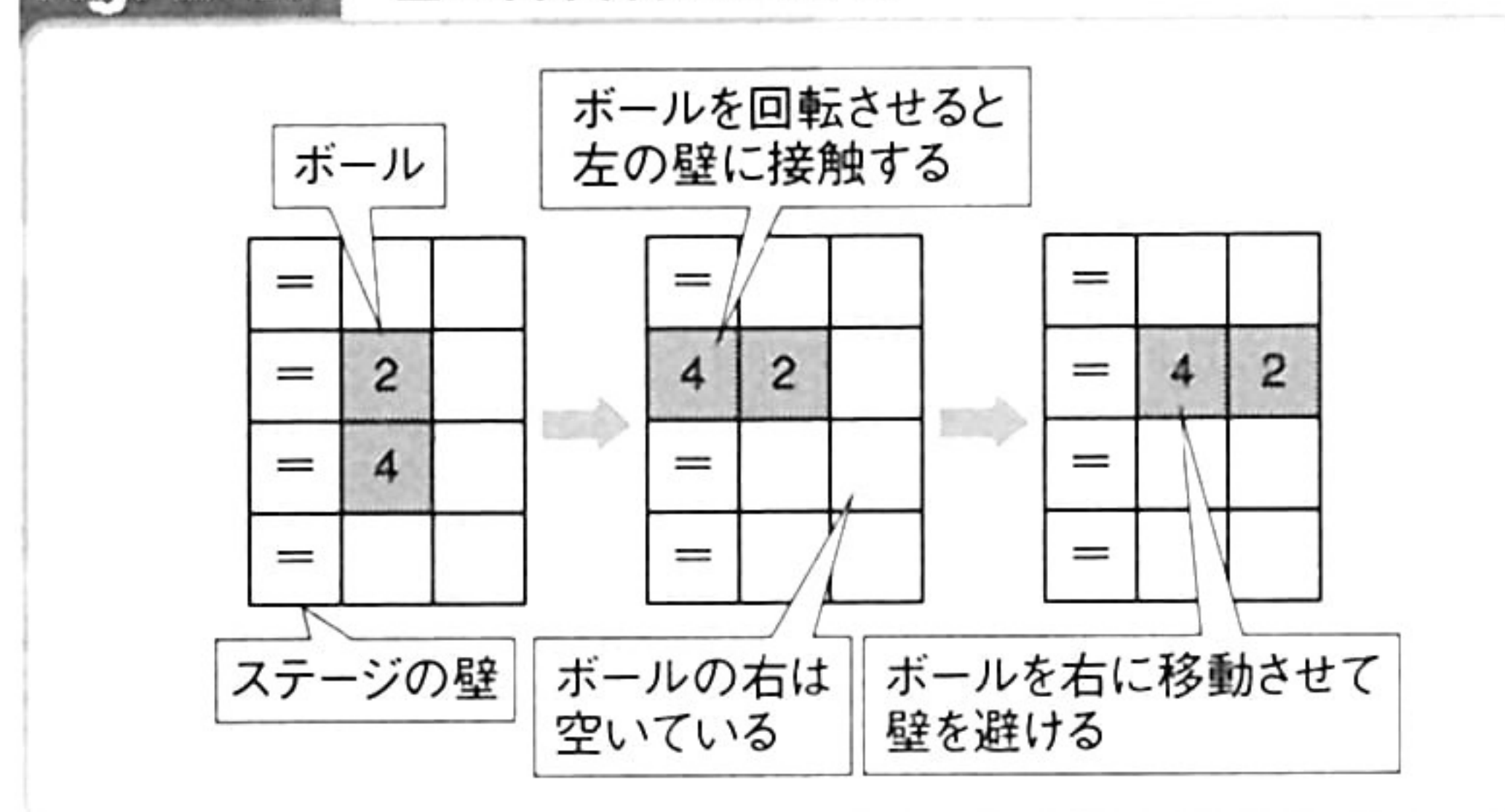



Fig. 2-67 左の障害物を避ける

ボールを回転させたら、ステージの壁や他のボールに接触するかどうかを調べます。そして、例えば右の障害物に接触する場合で、ボールの左が空いているときには、ボールを左に移動させます (Fig. 2-66)。

左の障害物に接触する場合も同様です。ボールの右が空いているときには、ボールを右に移動させて、障害物を避けます (Fig. 2-67)。

下に障害物がある場合の処理

ボールの下に障害物がある場合についても考えましょう。この場合、ボールの上が空いているならば、ボールを上に移動させればよさそうに思えます (Fig. 2-68)。

しかし、ボールを上に移動するのは問題があります。この動きを利用すると、無限にボールを空中に浮かせることができてしまいます (Fig. 2-69)。落ち物パズルゲームでは、物体はいつかは地面に落ちるのが基本なので、これはゲームバランスを崩す原因になります。特に『ぷよぷよ』のような対戦ゲームでは、いつまでも勝負がつかなくなる危険があります。

そこでボールを上に移動するのではなく、一時的に「3×3」のセルのなかでボールを上に寄

Fig. 2-68 ボールを上に移動させる

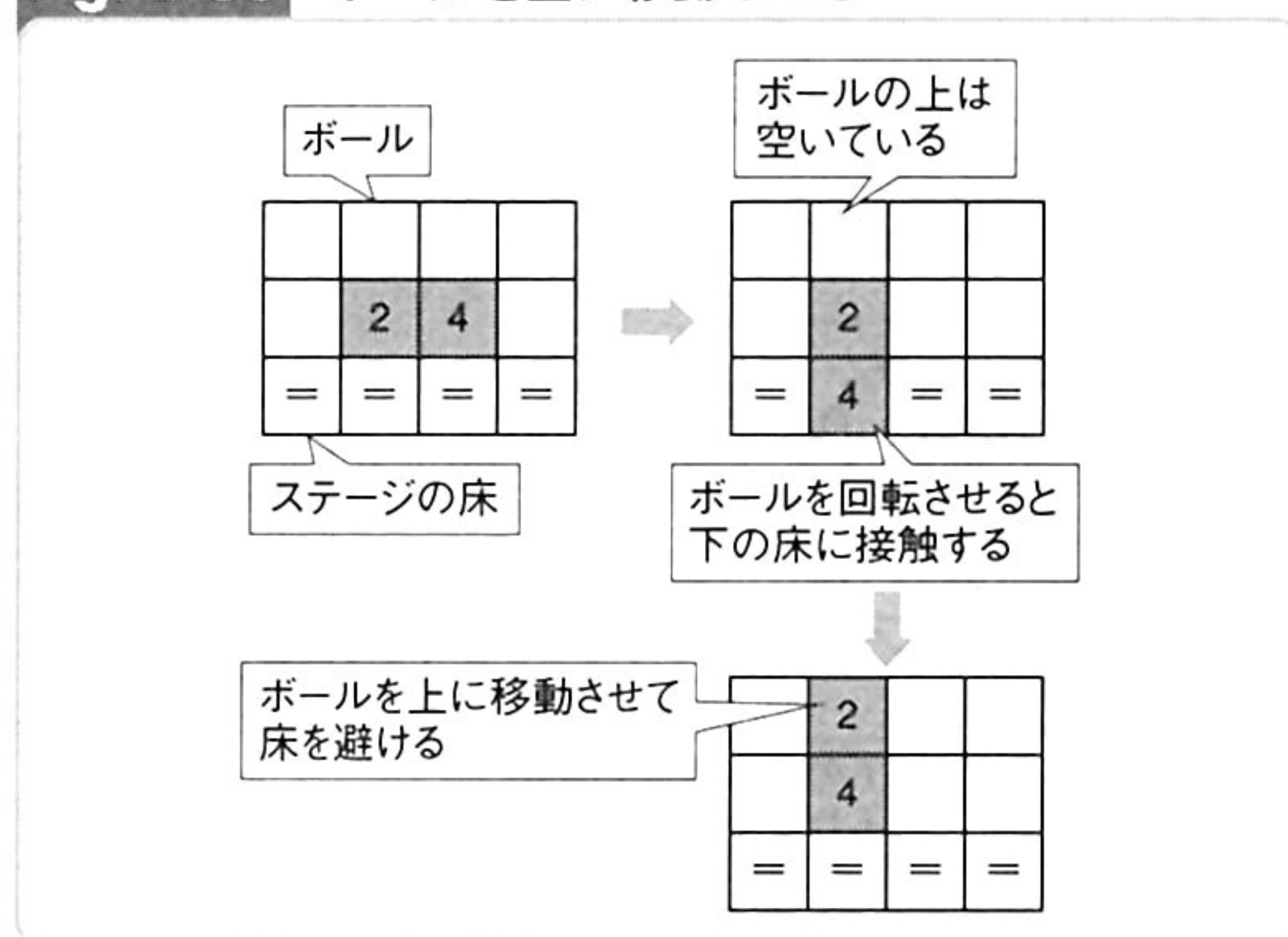



Fig. 2-69 ボールを無限に空中に浮かせる

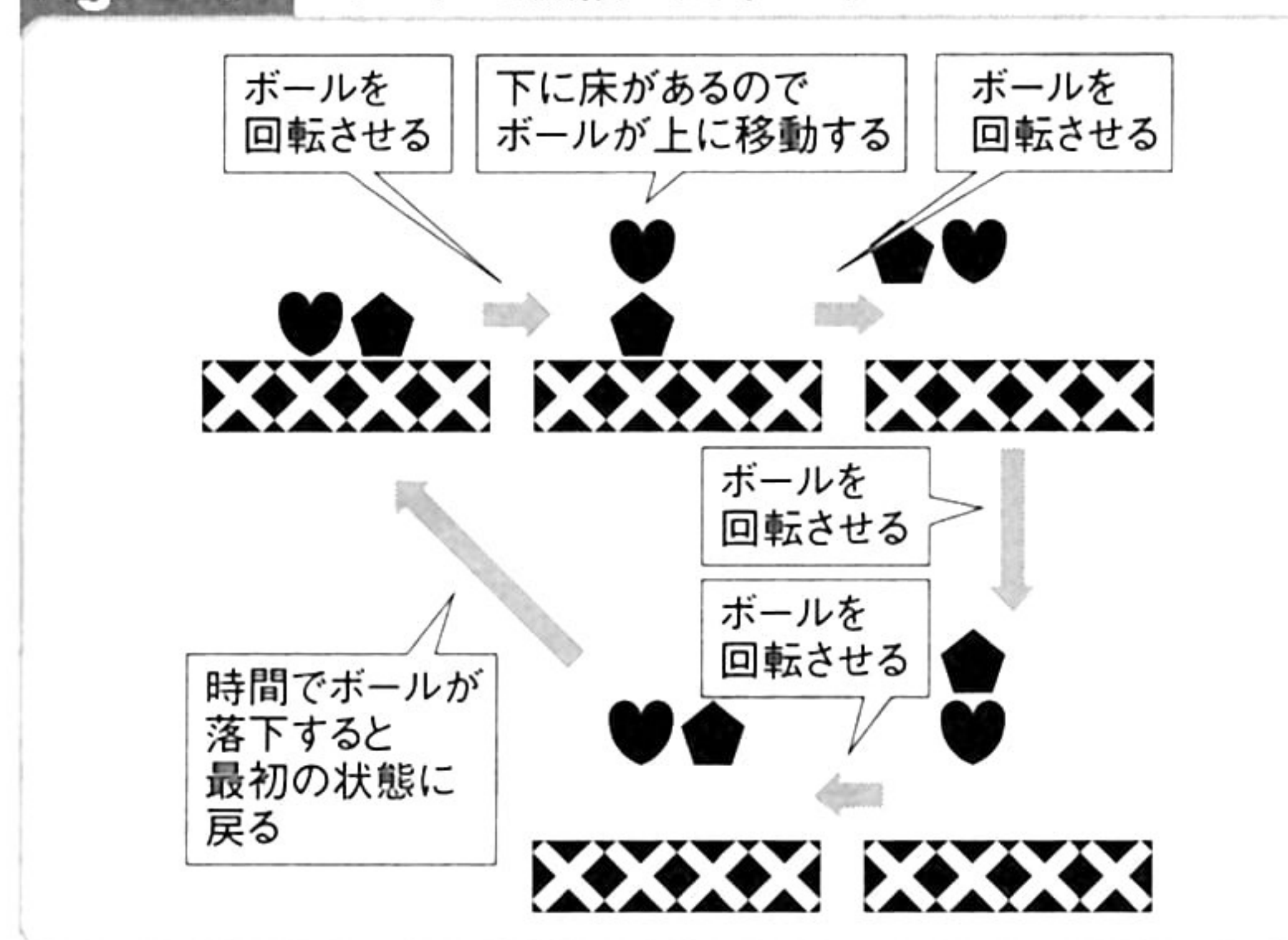
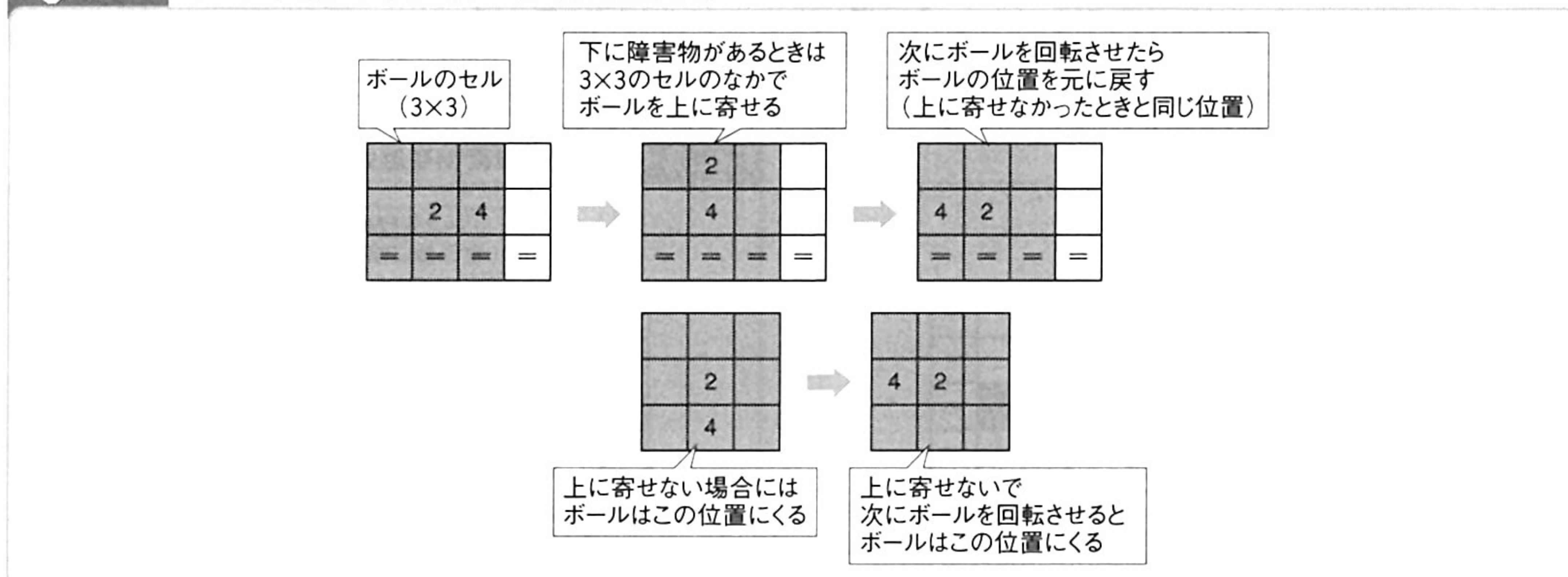



Fig. 2-70 3×3のセルのなかでボールを上に寄せる

せます (Fig. 2-70)。そして、次にボールを回転させたときには、ボールの位置を元に戻します。これで、ボールが無限に浮くことはなく、いつかは着地するようになります。

プログラム

List 2-11はボールを回転させるプログラムです。ボールの移動処理を掲載しました。

ボールを回転させる処理は、落下や左右への移動とともに、入力状態で行います。ボタンの入力を検出したら、ボールのセルを操作して、回転させます。

ボールを回転させたときに、ステージの壁や床、あるいは他のボールに接触したら、障害物の回避を試みます。左右に障害物があるときには、左右にボールを移動させます。下に障害物があるときには、セルのなかでボールを上に寄せます。上に寄せたときには、次にボールを回転させる前に、ボールを下に寄せて、元の状態に戻しておきます。

ボールの回転は、セルを入れ替えることによって行います。セルを入れ替える順番を逆にすると、ボールは逆方向に回転します。サンプルではこれを利用して、障害物が回避できないときに、ボールの回転を取り消しています。

List 2-11 障害物を避けながらボールを回転させる (CDroppingBallBallクラス)

```
// ボールの移動処理
bool CDroppingBallBall::Move(const CInputState*) {

    // ... (中略) ...

    // 入力状態
    if (State==0) {

        // ... (中略) ...

        // ボタンを入力しており、
        // かつ直前にボタンを離していたら、ボールを回転させる
        if (is->Button[0] && !PrevButton) {

            // セルのなかでボールを上に寄せていた場合には、
            // 回転に先立って、ボールを下に戻す
            if (Turn==2 && CurrentBall->Get(1, 2)==' ') {

                // 下と中央のセルを入れ替える
                CurrentBall->Swap(1, 2, 1, 1);

                // 中央と上のセルを入れ替える
                CurrentBall->Swap(1, 1, 1, 0);
            }

            // 回転角度 (0～3) を更新する
```

```
Turn=(Turn+1)%DROPPING_BALL_TURN_COUNT;

// ボールを回転させる
// 上と左、左と下、下と右、の順にセルを入れ替える
CurrentBall->Swap(1, 0, 0, 1);
CurrentBall->Swap(0, 1, 1, 2);
CurrentBall->Swap(1, 2, 2, 1);

// 回転後のボールがステージに接触したときの処理
if (StageCell->Hit(CX, CY, CurrentBall)) {

    // 右に障害物があり、左が空いているならば、左に避ける
    if (
        Turn==1 &&
        !StageCell->Hit(CX-1, CY, CurrentBall)
    ) {
        CX--;
        X=CX;
    } else

    // 左に障害物があり、右が空いているならば、
    // 右に避ける
    if (
        Turn==3 &&
        !StageCell->Hit(CX+1, CY, CurrentBall)
    ) {
        CX++;
        X=CX;
    } else

    // 下に障害物があり、上が空いているならば、
    // セルのなかでボールを上に寄せる
    if (
        Turn==2 &&
        !StageCell->Hit(CX, CY-1, CurrentBall)
    ) {
        // 中央と上のセルを入れ替える
        CurrentBall->Swap(1, 1, 1, 0);

        // 下と中央のセルを入れ替える
        CurrentBall->Swap(1, 2, 1, 1);
    } else

    // 障害物が回避できなかったときには、回転を取り消す
    {
        // 回転角度を元に戻す
        Turn=
            (Turn-1+DROPPING_BALL_TURN_COUNT)%
            DROPPING_BALL_TURN_COUNT;
```

```
                    // ボールの回転を取り消す
                    // 最初に回転させたときとは逆に、
                    // 下と右、左と下、上と左、の順にセルを入れ替える
                    CurrentBall->Swap(1, 2, 2, 1);
                    CurrentBall->Swap(0, 1, 1, 2);
                    CurrentBall->Swap(1, 0, 0, 1);
                }
            }
        }

        // 直前のボタンの入力状態を保存する
        PrevButton=is->Button[0];

        // レバーを下に入れ続けたときに、ボールが次々に落ちないようにするための処理
        if (!is->Down) PrevDown=false;
    }

    // ...（中略）...
}
```

着地したボールが2つに分かれる

ボールが着地したときに、1組だったボールが2つに分かれて、別々に落ちていくアクションです。積み上げられたボールの隙間に、落としたい種類のボールだけを落とすことができるので、ボールをより自由自在に積むことができます。

ボールが2つに分かれるのは、段差のある地面に対して、ボールを横向きに落としたときです（Fig. 2-71）。着地すると、ボールは2つに分かれて、それぞれステージの床や他のボールに

Fig. 2-71 ボールを横向きに落とす

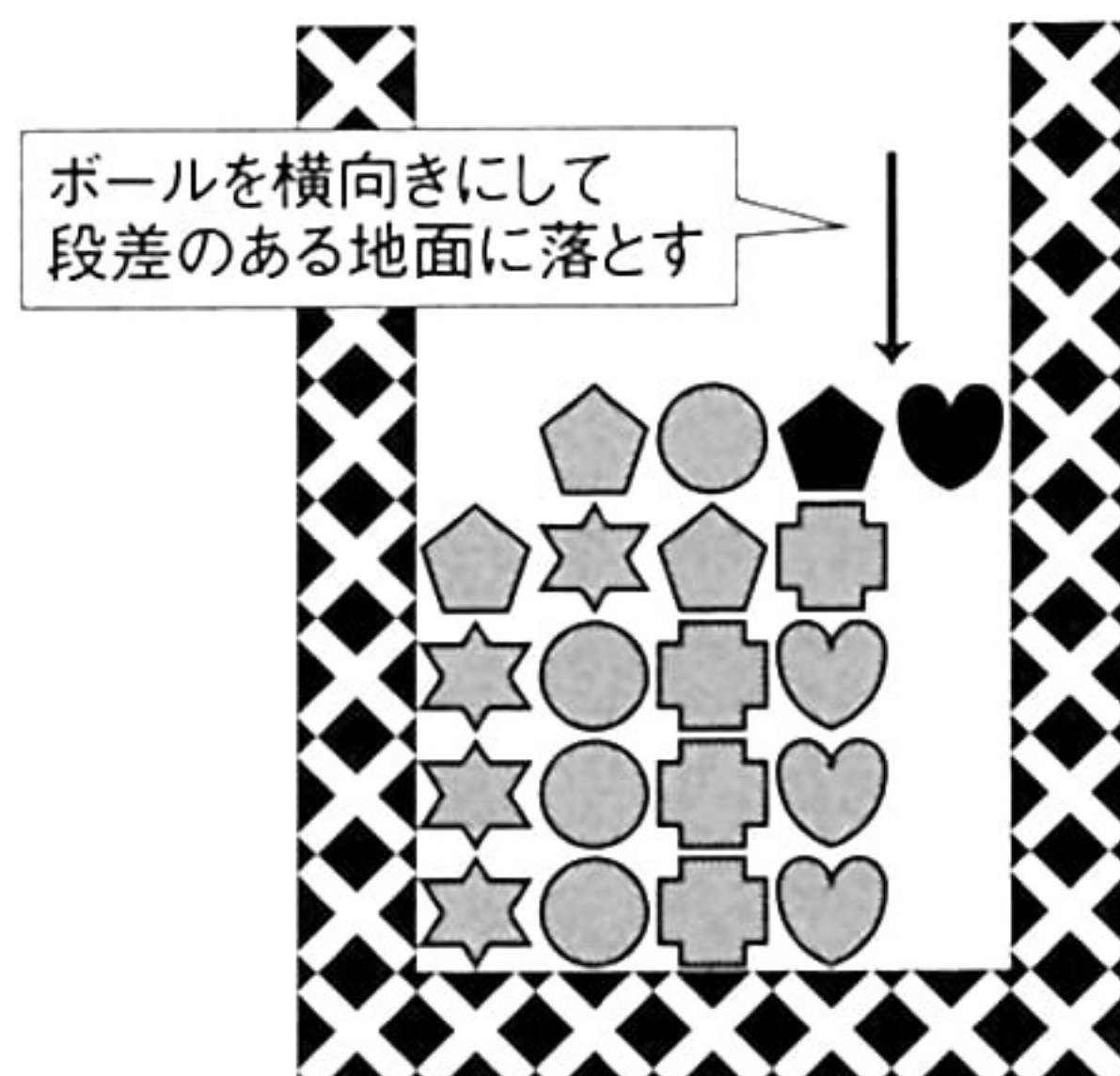
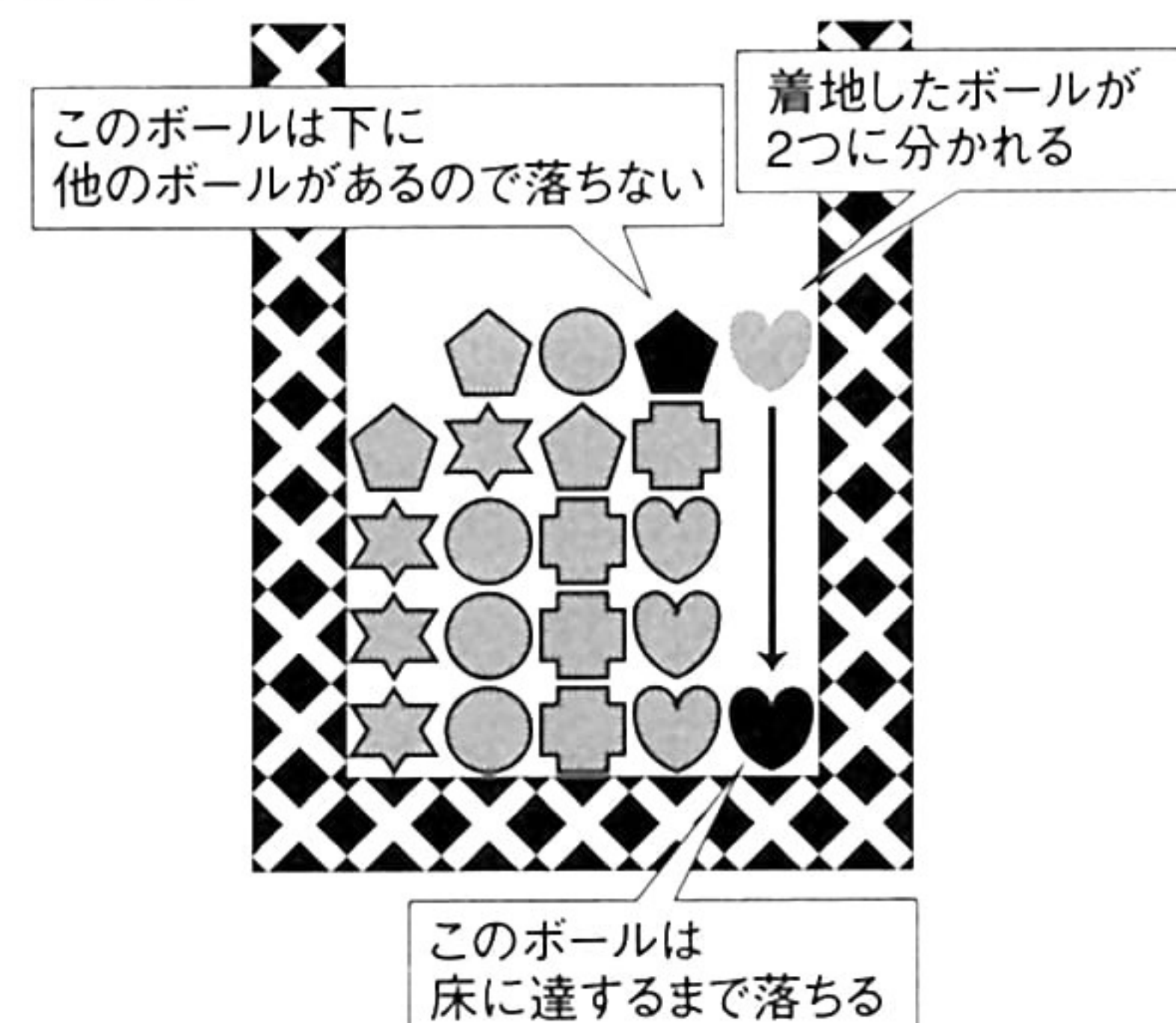



Fig. 2-72 ボールが2つに分かれて落ちる

着地するまで落下します（Fig. 2-72）。

アルゴリズム

ボールが2つに分かれて、別々に落ちていくアクションは、「縦横斜めに揃える」（→p. 82）で宝石を落とす処理に似ています。ボールを2つに分けて落とす処理というよりも、ステージ全体を調べて、下に何もないボールを落とす処理になります。

ボールが着地したら、ボールのセルをステージのセルに合成します。次に、ステージ全体のセルを調べて、下に何もないボールを探します（Fig. 2-73）。そして、そのボールを落ちるボールとしてマークします。

落ちるボールの上に、他のボールが載っている場合もあります（Fig. 2-74）。落ちるボールよりも上にあるボールは、下にあるボールが落ちれば、支えがなくなって下に落ちます。そこで、落ちるボールよりも上にあるすべてのボールも、落ちるボールとしてマークします。

落ちるボールは、それぞれ1段ずつ下に落とします（Fig. 2-75）。セル上ではボールを1段単位

Fig. 2-73 下に何もないボールを探す

=						=
=						=
=						=
=						=
=						=
=		4	0	4	2	=
=	4	1	4	3		=
=	1	0	3	2		=
=	1	0	3	2		=
=	1	0	3	2		=
=	=	=	=	=	=	=

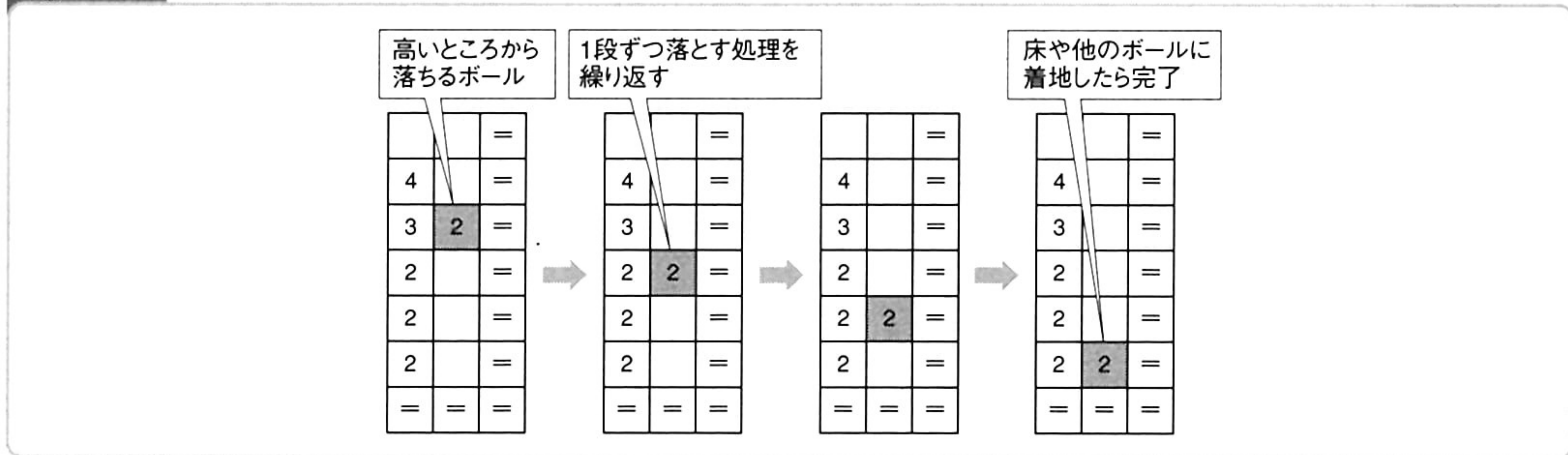
Fig. 2-74 落ちるボールの上にあるすべてのボールをマークする

=						=
=						=
=						=
=						=
=						=
=		4	0	4		=
=	4	1	4	3		=
=	1	0	3			=
=	1	0	3			=
=	1	0	3			=
=	=	=	=	=	=	=

Fig. 2-75 落ちるボールを1段落とす

=						=
=						=
=						=
=						=
=						=
=		4	0	4		=
=	4	1	4	3	2	=
=	1	0	3	2		=
=	1	0	3	2		=
=	1	0	3	2		=
=	=	=	=	=	=	=

Fig. 2-76 高いところから落ちるボール

高いところから落ちるボール
1段ずつ落とす処理を繰り返す
床や他のボールに着地したら完了

で落としますが、画面上では少しずつボールを動かして、ボールが滑らかに落ちるように描画します。ボールを落ちるボールとしてマークするのは、落ちるボールと落ちないボールを区別して、描画方法を変えるためです。

画面上でボールが1段落ちたら、落ちるボールのマークを解除します。そして再びステージ全体を調べて、落ちるボールがないかどうかを調べます。落ちるボールがあれば、前回と同様の手順でボールを落とします。

落ちるボールがあるかぎり、この手順を繰り返します。高いところから落ちるボールは、1段ずつ落とす処理を何度も繰り返すことによって、着地させます（Fig. 2-76）。

プログラム

List 2-12は着地したボールが2つに分かれるプログラムです。ボールの移動処理と描画処理を掲載しました。

移動処理は、落下判定状態と落下状態に分かれています。ボールが着地すると、落下判定状態に移行します。落下判定状態では、ステージ全体を調べて、下に何もないボールを落ちるボールとしてマークします。落ちるボールの上にあるすべてのボールも、落ちるボールとしてマークします。そして、マークしたボールを1段ずつ下に落とします。

落ちるボールがある場合には、落下判定状態から落下状態に移ります。落下状態では、一定時間が経過するのを待って、落ちるボールのマークを解除します。そして、落下判定状態に戻ります。

落ちるボールのマークは、セルの最上位ビットをセットすることで行います。これは別の方法で行ってもかまいません。例えば、セルの種類を「0～4」から「A～E」に置き換える、といった方法も使えます。

描画処理では、ステージ内のすべてのセルを描画します。落ちるボールについては、タイマーを使って少しずつ座標を変え、画面上で滑らかに移動するように描画します。

List 2-12 着地したボールが2つに分かれる(CDroppingBallBallクラス)

```
// ボールの移動処理
bool CDroppingBallBall::Move(const CInputState*) {

    // ...(中略)...

    // 落下判定状態
    if (State==2) {

        // 落ちるボールがまったくない場合には、消去判定状態へ移行する
        State=4;

        // 相手の攻撃を受けている場合には、攻撃ボールを発生させる
        Attack();

        // ステージ全体のセルを下から上に調べる
        for (int x=FieldLeft; x<FieldRight; x++) {
            for (
                int y=DROPPING_BALL_FIELD_BOTTOM-1;
                y>=DROPPING_BALL_FIELD_TOP;
                y--
            ) {
                // 下に何もないボールを探す
                if (
                    StageCell->Get(x, y)==' ' &&
                    StageCell->Get(x, y-1)!=' '
                ) {
                    // 下に何もないボールがあったら、
                    // そのボールと、それより上にあるすべてのボールを、
                    // 落ちるボールとしてマークする
                    // (セルの最上位ビットをセットする)
                    for (; y>=DROPPING_BALL_FIELD_TOP; y--) {

                        // セルがボールかどうかを調べ、
                        // ボールの場合には落ちるボールとしてマークし、
                        // 1段ずつ下に移動させる
                        char c=StageCell->Get(x, y-1);
                        if (c!=' ') {
                            StageCell->Set(x, y, c|0x80);
                        } else {
                            StageCell->Set(x, y, ' ');
                        }
                    }

                    // タイマーを設定し、落下状態に移行する
                    Time=0;
                    State=3;
                }
            }
```

```
        }
    }

    //落下状態
    if (State==3) {

        //タイマーの更新
        Time++;

        //タイマーが一定値に達したら、落ちるボールのマークを解除する
        if (Time==10) {

            //ステージ上のすべてのセルについて処理する
            for (
                int y=DROPPING_BALL_FIELD_TOP;
                y<DROPPING_BALL_FIELD_BOTTOM;
                y++
            ) {
                for (int x=FieldLeft; x<FieldRight; x++) {

                    //落ちるボールのマークを解除する
                    //(セルの最上位ビットをクリアする)
                    StageCell->Set(
                        x, y, StageCell->Get(x, y)&0x7f);
                }
            }

            //落下判定状態に移行する
            //ボールは1段ずつ落下するため、
            //高いところから落ちるボールが着地するまでには、
            //落下判定状態と落下状態を何度も繰り返す必要がある
            State=2;
        }
    }

    //...(中略)...
}

//ボールの描画処理
void CDroppingBallBall::Draw() {

    //画面の解像度を取得する
    float
        sw=Game->GetGraphics()->GetWidth()/MAX_X,
        sh=Game->GetGraphics()->GetHeight()/MAX_Y;

    //...(中略)...

    //ステージのすべてのセルについて処理する
```

```
    for (
        int y=DROPPING_BALL_FIELD_TOP;
        y<DROPPING_BALL_FIELD_BOTTOM;
        y++
    ) {
        for (int x=FieldLeft; x<FieldRight; x++) {

            // セルの種類を取得する
            char c=StageCell->Get(x, y);

            // 落ちるボール以外のボール
            if (
                '0'<=c &&
                c<'0'+DROPPING_BALL_COLOR_COUNT*2
            ) {
                // ボールの種類に応じたグラフィックを描画する
                Game->Texture[TEX_BALL0+(c-'0')]->Draw(
                    x*sw, y*sh, sw, sh, 0, 0, 1, 1, COL_BLACK
                );
            } else

            // 落ちるボール
            if (c&0x80) {

                // タイマーを使って座標を少しずつ変化させ、
                // ボールが滑らかに落ちるように描画する
                Game->Texture[TEX_BALL0+((c&0x7f)-'0')]->Draw(
                    x*sw, (y-1+Time*0.1f)*sh,
                    sw, sh, 0, 0, 1, 1, COL_BLACK
                );
            } else

            // ...（中略）...
        }
    }
}
```

連鎖的に消す

同じ種類のボールを規定数以上隣接させると、ボールが消えるアクションです。ボールが消えると、下に何もなくなったボールが落ちます。ボールが落ちた後には、再び同じ種類のボールが隣接して、連鎖的に消えることがあります。

ボールが着地したときに、同じ種類のボールが一定数以上隣接していたら、それらのボールが

消えます（Fig. 2-77）。ここでは4個以上のボールが隣接したときに消えることにしましょう。

ボールが縦または横に並んでいたら、隣接しているとします。斜めに並んでいても、隣接していることにはしません。

ボールが消えると、上にあったボールが落ちてきます（Fig. 2-78）。このとき、再び同じ種類のボールが隣接すると、ボールが消えます（Fig. 2-79）。

ボールが消えると、上にあったボールが落ち、再びボールが隣接すれば消え、また上にあるボールが落ち……といった具合に、ボールは連鎖的に消えていきます。ボールが消える過程を予想しながら上手に積み上げると、大きな連鎖を作ることができます（Fig. 2-80）。このように連鎖を意図してボールを積み上げることを、「連鎖を組む」と呼びます。

『ぷよぷよ』では、素早く大きな連鎖を組むことが必勝法です。4連鎖・5連鎖程度は当たり前で、作品によっては10連鎖以上の長い連鎖が要求されるものもあります。連鎖を組む方法としては、ボールの落差を利用して必要なボールを上に積んでおく方法（階段積み）や、同じ種類のボールを他のボールで仕切っておく方法（はさみ込み）など、いろいろな手法が研究されています。

Fig. 2-77　同じ種類のボールが隣接する

同じ種類のボールが
規定数（4個）以上隣接すると
ボールが消える

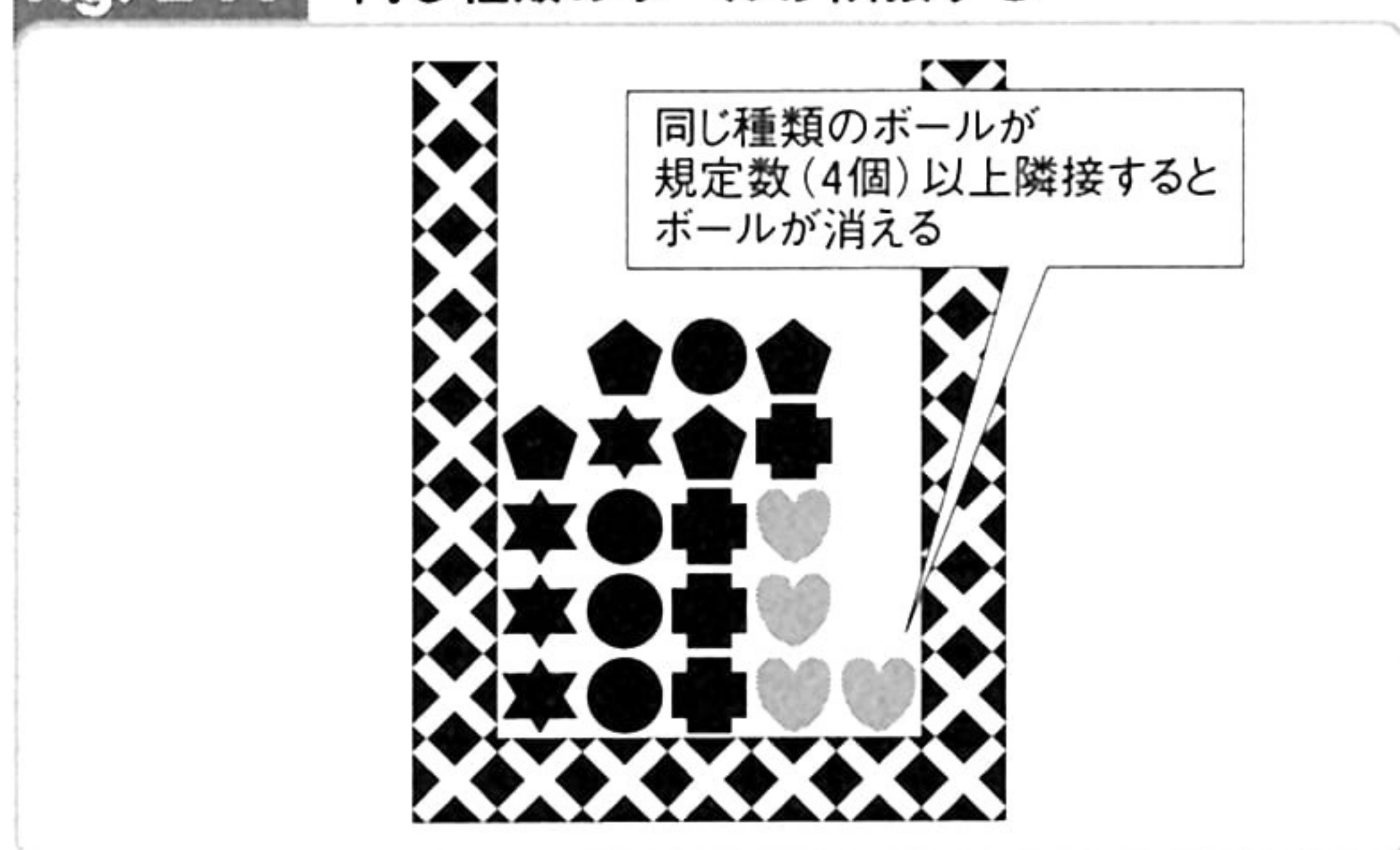
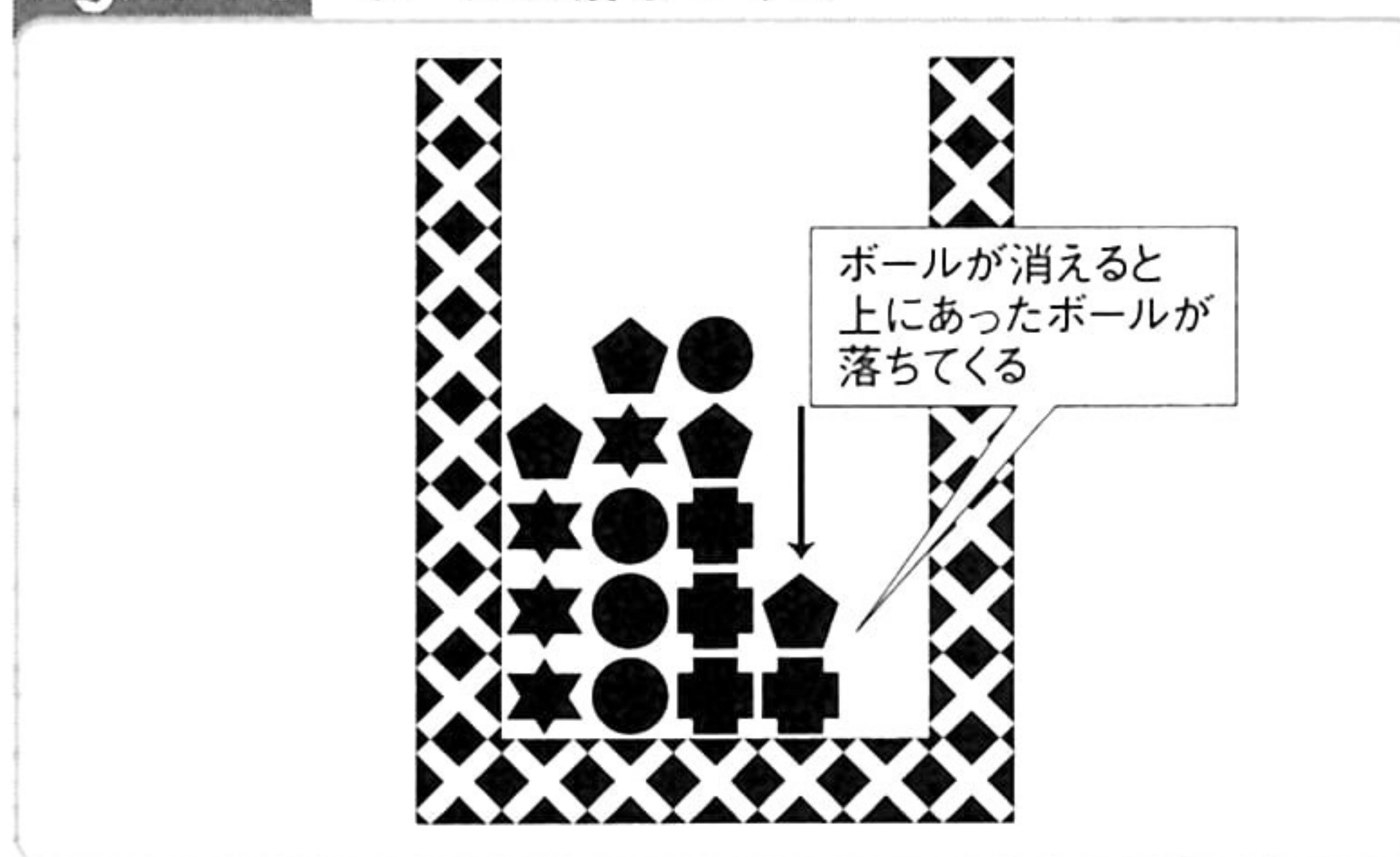
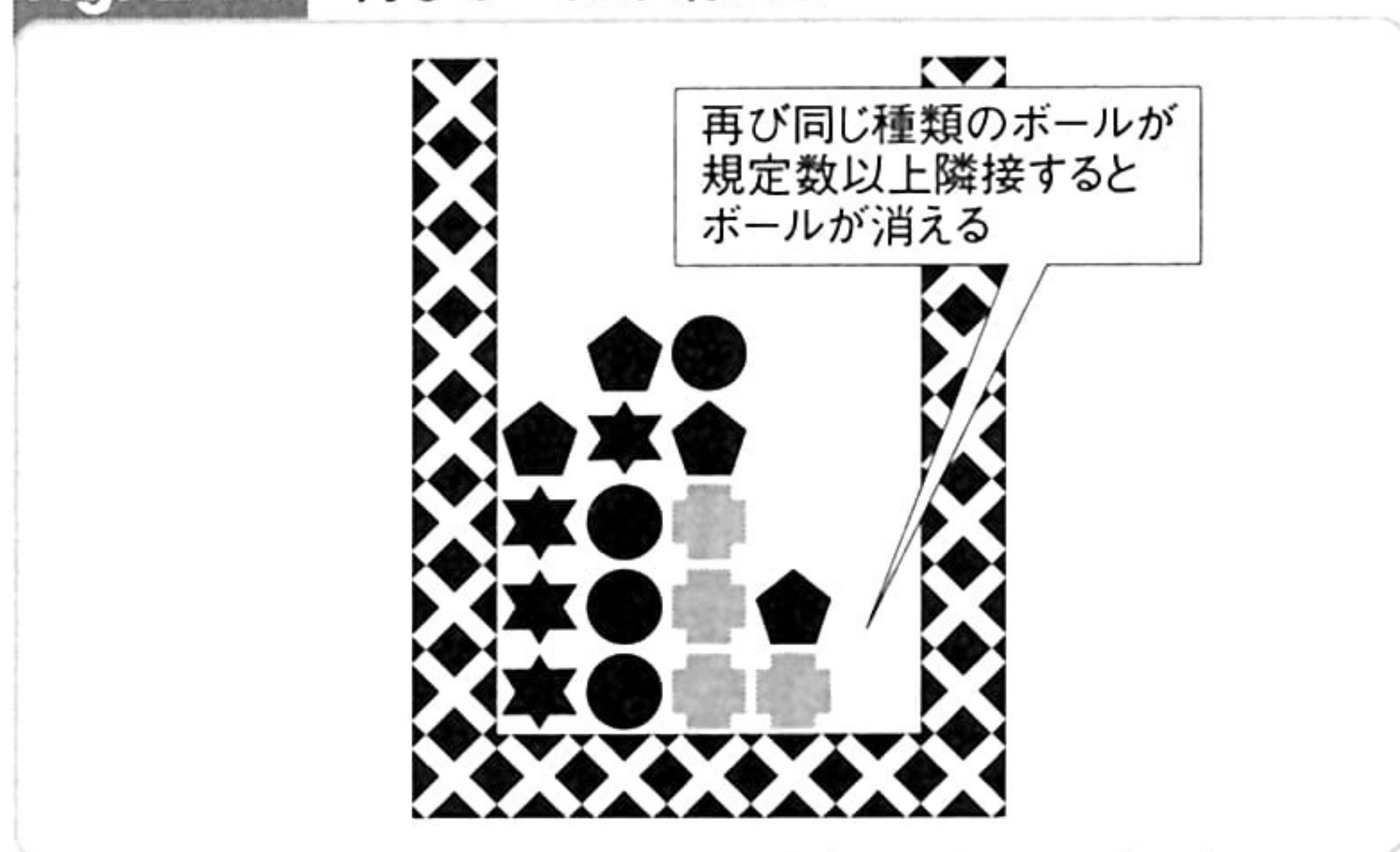
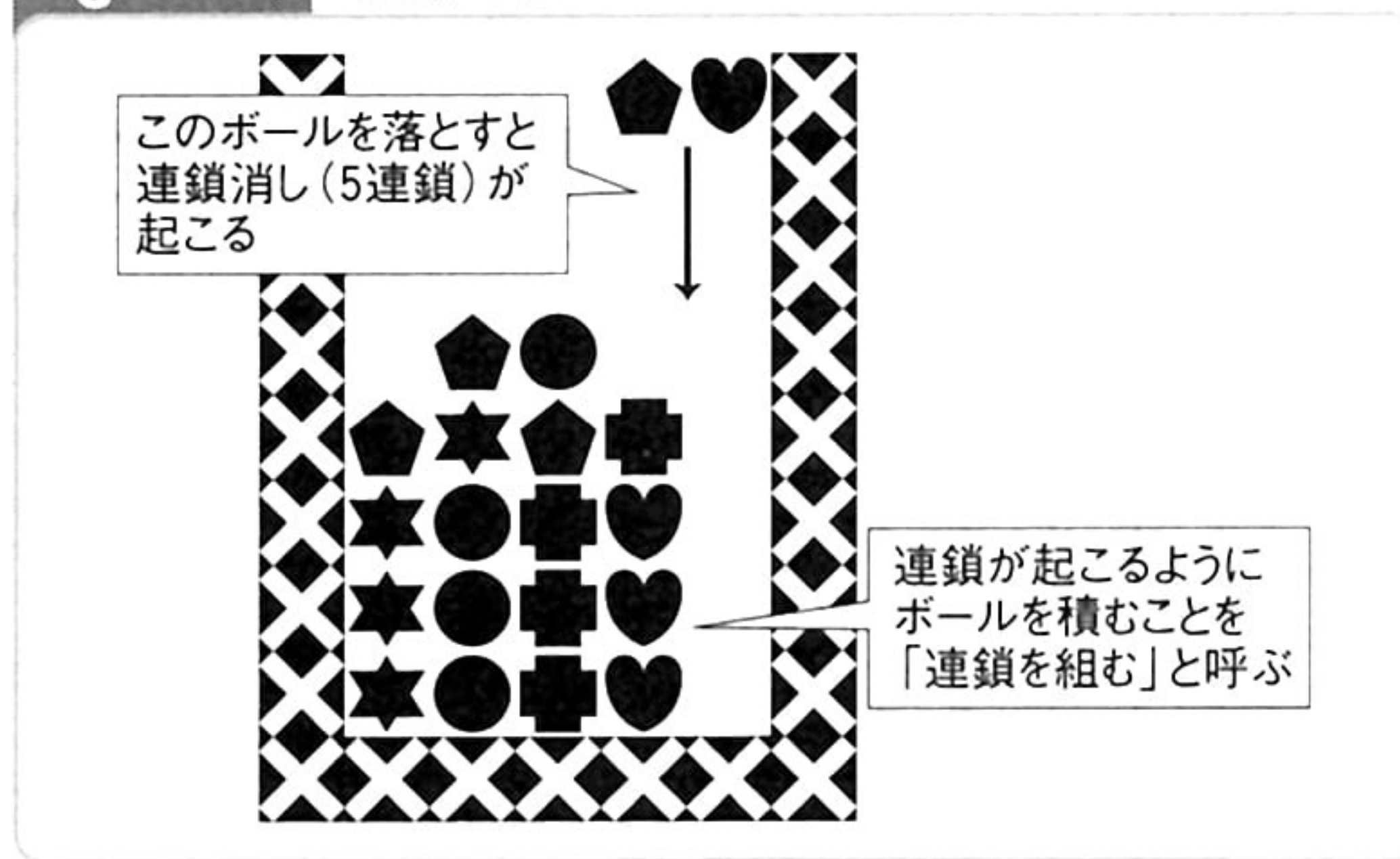
Fig. 2-78　ボールが落ちてくる

ボールが消えると
上にあったボールが
落ちてくる

Fig. 2-79　再びボールが消える

再び同じ種類のボールが
規定数以上隣接すると
ボールが消える

Fig. 2-80　連鎖を組む

このボールを落とすと
連鎖消し（5連鎖）が
起こる

連鎖が起こるように
ボールを積むことを
「連鎖を組む」と呼ぶ

アルゴリズム

ボールを消すには、ボールが着地したときに、ステージのセルを調べます。「ブロックを1段揃えて消す」(→p. 64) や「縦横斜めに揃える」(→p. 82) に似ていますが、隣接したボールを数えるために再帰的な処理が必要なことが違います。

ボールが消えるのは、規定数以上の同じ種類のボールが縦または横に隣接しているときです。ここでは4個以上が隣接したときに消えるとします。3個や5個が隣接したときに消えるルールにする場合も、処理の方法はまったく同じです。

ステージのすべてのセルについて、順番に処理します。ボールのセルを見つけたら、そのボールの上下左右を調べて、同じ種類のボールを探します (Fig. 2-81)。

同じ種類のボールがあったら、さらにそのボールの上下左右を調べて、同じ種類のボールを探します (Fig. 2-82)。同じ種類のボールが見つかるかぎり、この手順を繰り返します。ただし、一度調べたセルは、二度は調べません。

Fig. 2-81 隣接する同じ種類のボールを探す

=						=
=						=
=						=
=						=
=						=
=		4	0	4		=
=	4	1	4	3		=
=	1	0	3	2		=
=	1	0	3	2		=
=	1	0	3	2	2	=
=	=	=	=	=	=	=

このボールについて
上下左右のセルを調べる

下に同じ種類の
ボールがある

Fig. 2-82 同じ種類のボールが見つかるかぎり探す

=						=
=						=
=						=
=						=
=						=
=		4	0	4		=
=	4	1	4	3		=
=	1	0	3	2		=
=	1	0	3	2		=
=	1	0	3	2	2	=
=	=	=	=	=	=	=

このセルはすでに調べたので
もう調べない

このボールについて
上下左右のセルを調べる

さらに下にも
同じ種類のボールがある

Fig. 2-83 隣接した同じ種類のボールをすべて見つける

=						=
=						=
=						=
=						=
=						=
=		4	0	4		=
=	4	1	4	3		=
=	1	0	3	2		=
=	1	0	3	2		=
=	1	0	3	2	2	=
=	=	=	=	=	=	=

このボールについて
上下左右のセルを調べる

右に同じ種類の
ボールがある

隣接した同じ種類のボールを
すべて見つけるまで続ける

手順を繰り返すと、隣接している同じ種類のボールをすべて見つけることができます（Fig. 2-83）。見つけたボールの数が規定数（4個）以上ならば、それらのボールを消えるボールとしてマークします（Fig. 2-84）。

ボールをマークすると、これから消えるボールと、消えないボールとを区別することができます。ボールを描画するときに、消えるボールについては時間とともにだんだん薄くなるように表示すると、どのボールが消えたのかがわかりやすくなります。

一定時間が経過したら、消えるボールを完全に消します（Fig. 2-85）。ボールが消えると、上にあったボールは下に落ちます。これは「着地したボールが2つに分かれる」（→p. 101）の段階に戻って処理します。

ボールに関する処理の状態遷移をまとめたものがFig. 2-86です。下に何もないボールを落とす状態（落下判定状態・落下状態）と、隣接したボールを消す状態（消去判定状態・消去状態）

Fig. 2-84　消えるボールをマークする

=						=
=						=
=						=
=						=
=						=
=		4	0	4		=
=	4	1	4	3		
=	1	0	3	2		
=	1	0	3	2		=
=	1	0	3	2	2	=
=	=	=	=	=	=	=

隣接した同じ種類のボールが
規定数以上ならば
消えるボールとしてマークする

Fig. 2-85　ボールを消す

=						=
=						=
=						=
=						=
=						=
=		4	0			=
=	4	1	4	↓		=
=	1	0	3	↓		=
=	1	0	3	4		=
=	1	0	3	3		=
=	=	=	=	=	=	=

消えたボールの
上にあったボールが
下に落ちる

Fig. 2-86　ボールの処理に関する状態遷移

入力状態
ボールを左右に動かす
ボールを回転させる
ボールを落下させる

ボールが床や他のボールに着地した

左右に移動した
または落下した

移動状態
タイマーを使って
ボールを滑らかに動かす

落下判定状態
下に何もないボールがあったら
落ちるボールとしてマークする

落ちるボールがなかった

落ちるボールがあった

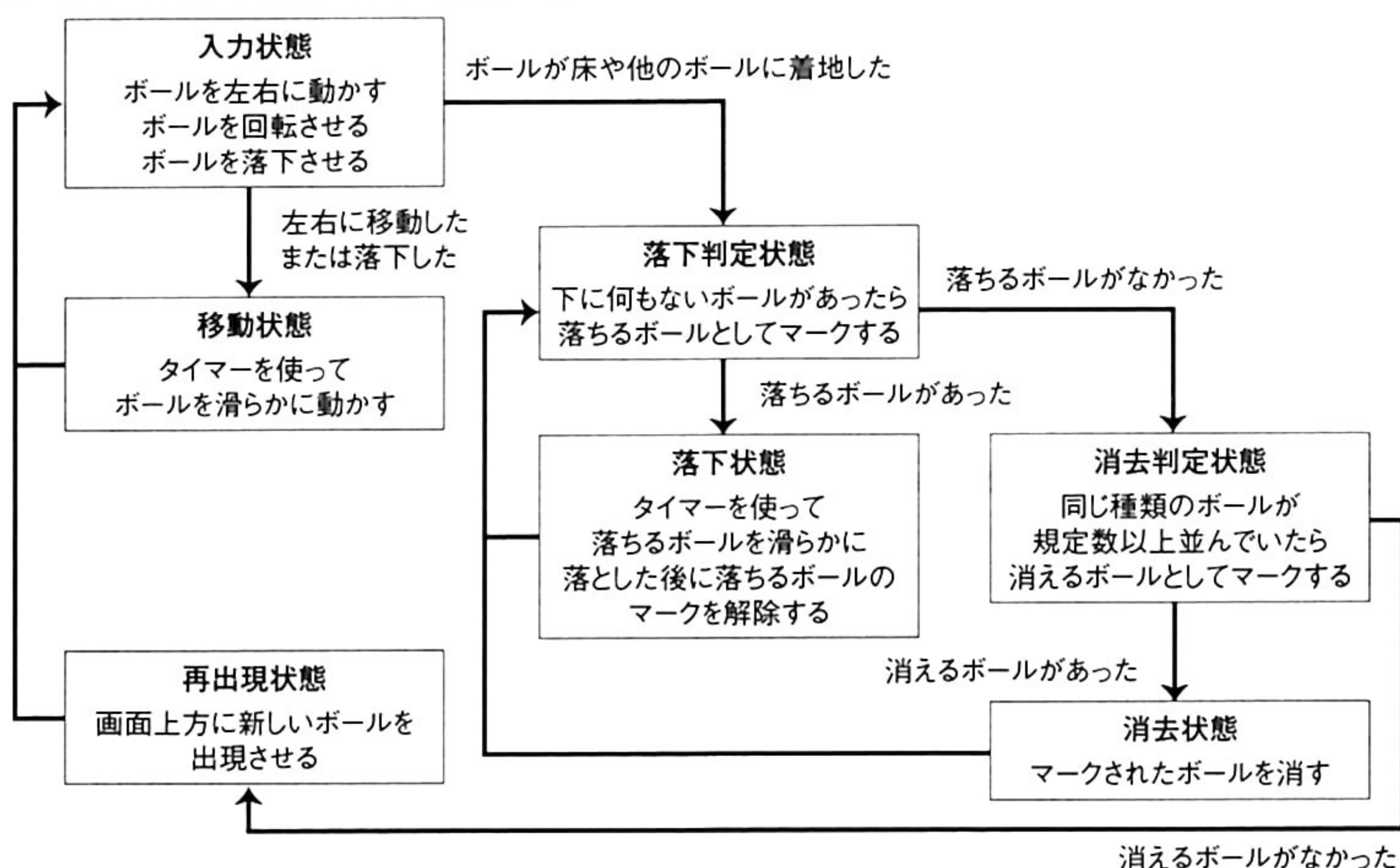
落下状態
タイマーを使って
落ちるボールを滑らかに
落とした後に落ちるボールの
マークを解除する

消去判定状態
同じ種類のボールが
規定数以上並んでいたら
消えるボールとしてマークする

消えるボールがあった

再出現状態
画面上方に新しいボールを
出現させる

消去状態
マークされたボールを消す

消えるボールがなかった

の間がループになっていて、条件を満たすかぎり何度も繰り返すことがポイントです。これで、ボールが消えては上のボールが落ち、再びボールが消えて……という連鎖が起こります。

プログラム

List 2-13はボールを連鎖的に消すプログラムです。ボールの移動処理と、隣接したボールを数える処理、隣接したボールを消えるボールとしてマークする処理、消えるボールを消す処理、そしてボールの描画処理を掲載しました。

移動処理（Move関数）は消去判定状態・消去状態・再出現状態に分かれています。消去判定状態では、隣接している同じ種類のボールを数えて、規定数以上ならば消えるボールとしてマークします。消去状態では、タイマーが一定値になるのを待ってから、消えるボールを消します。再出現状態では、ステージの上方に新しいボールを出現させます。

隣接したボールを数える処理（Count関数）では、再帰呼び出し（関数からその関数自体を呼び出すこと）を使います。同じ種類のボールが見つかるかぎり、上下左右のセルについても処理を再帰的に行うことによって、隣接したすべてのボールを見つけ出します。

隣接したボールを消えるボールとしてマークする処理（BeginErasing関数）でも、再帰呼び出しを使います。同じ種類のボールが見つかるかぎり、上下左右のセルについても再帰的に処理を行い、隣接したすべてのボールを消えるボールにします。

消えるボールを消す処理（EndErasing関数）では、消えるボールのセルを空にすることによって、ボールを完全に消します。ボールをマークしておいて、一定時間が経過してから実際に消すのは、ボールが消える様子を表示するためです。描画処理（Draw関数）では、消えるボールと消えないボールを区別して、消えるボールについては時間とともにだんだん消えていくように表示しています。

カウントずみのボールをマークするときには、セルの最上位ビットをセットしています。消えるボールをマークするときには、上から2番目のビットをセットしています。セルが特別な状態にあることを示すためには、このようにビット演算を使うのが便利ですが、他の方法を使ってもかまいません。

ボールを消したときには、消した数や連鎖数に応じて、相手側のフィールドに攻撃ボールを送り込みます。この処理については「相手側にボールを降らせる」（→p. 115）で詳しく解説します。

List 2-13　連鎖的に消す（CDroppingBallBallクラス）

```
// ボールの移動処理
bool CDroppingBallBall::Move(const CInputState*) {

    // ...（中略）...

    // 消去判定状態
    if (State==4) {
```

```
//連鎖カウント(連鎖消しの回数)を増やす
ChainCount++;

//ボールがまったく消えなかった場合には、
//再出現状態に移行する
State=6;

//ステージ内のすべてのセルについて、
//消えるボールがあるかどうかを調べる
for (
    int y=DROPPING_BALL_FIELD_TOP;
    y<DROPPING_BALL_FIELD_BOTTOM;
    y++
) {
    for (int x=FieldLeft; x<FieldRight; x++) {

        //セルの種類を取得する
        char c=StageCell->Get(x, y);

        //セルがカウントずみではなく、
        //かつ通常のボール(0〜4)ならば、
        //カウントを始める
        if (
            !(c&0x80) &&
            '0'<=c &&
            c<'0'+DROPPING_BALL_COLOR_COUNT
        ) {
            //同じ種類のボールがいくつ隣接しているのかを数える
            int count=Count(x, y, c);

            //隣接数が規定数(4個)以上ならば、
            //ボールを消す
            if (count>=ErasingCount) {

                //相手側の攻撃ボール数を増やす
                Opponent->AttackCount+=
                    (count-ErasingCount+1)*ChainCount;

                //消えるボールとしてマークする
                BeginErasing(x, y, c|0x80);

                //タイマーを設定し、
                //消去状態へ移行する
                Time=0;
                State=5;
            }
        }
    }
```

```
        }

        // ステージ内のすべてのボールについて、
        // カウントずみのマークを解除する
        for (
            int y=DROPPING_BALL_FIELD_TOP;
            y<DROPPING_BALL_FIELD_BOTTOM;
            y++
        ) {
            for (int x=FieldLeft; x<FieldRight; x++) {

                // カウントずみのマークを解除する
                // (最上位ビットをクリアする)
                StageCell->Set(x, y, StageCell->Get(x, y)&0x7f);
            }
        }
    }

    // 消去状態
    if (State==5) {

        // タイマーの更新
        Time++;

        // タイマーが一定値に達したら、
        // マークされたボールを消去する
        if (Time==10) {

            // ステージ内のすべてのセルについて処理する
            for (
                int y=DROPPING_BALL_FIELD_TOP;
                y<DROPPING_BALL_FIELD_BOTTOM;
                y++
            ) {
                for (int x=FieldLeft; x<FieldRight; x++) {
                    EndErasing(x, y);
                }
            }

            // 落下判定状態へ移行する
            // ボールが消えると、
            // 下に何もなくなったボールが落ちることがあるため
            State=2;
        }
    }

    // 再出現状態
    if (State==6) {
```

```
        //新しいボールをステージ上方に出現させる
        Init();
    }

    return true;
}

//隣接したボールを数える処理
//再帰的に呼び出す
int CDroppingBallBall::Count(int x, int y, char ball) {

    //セルの種類を取得する
    char c=StageCell->Get(x, y);

    //セルが同じ種類のボールのときの処理
    if (c==ball) {

        //ボールにカウントずみのマークを付ける
        StageCell->Set(x, y, c|0x80);

        //上下左右に隣接するセルについても、
        //同じ種類のボールが隣接しているかどうかを再帰的に調べ、
        //隣接しているボールの数の合計値を返す
        return
            1+
            Count(x-1, y, c)+
            Count(x+1, y, c)+
            Count(x, y-1, c)+
            Count(x, y+1, c);
    }

    //セルが同じ種類のボールではないときには、
    //隣接しているボールの数を0個として返す
    return 0;
}

//隣接したボールを消えるボールとしてマークする処理
//再帰的に呼び出す
void CDroppingBallBall::BeginErasing(int x, int y, char ball) {

    //セルの種類を取得する
    char c=StageCell->Get(x, y);

    //セルが同じ種類のボールのときの処理
    if (c==ball) {

        //ボールを消えるボールとしてマークする
        StageCell->Set(x, y, c|0x40);
```

```
			// 上下左右に隣接するセルについても、
			// 同じ種類のボールを消す処理を再帰的に行う
			BeginErasing(x-1, y, c);
			BeginErasing(x+1, y, c);
			BeginErasing(x, y-1, c);
			BeginErasing(x, y+1, c);
		} else

		// セルが攻撃ボールのときの処理
		if (c=='0'+DROPPING_BALL_COLOR_COUNT) {

			// ボールを消えるボールとしてマークする
			StageCell->Set(x, y, c|0x40);

			// 相手の攻撃ボール数を増やす
			Opponent->AttackCount+=ChainCount;
		}
}

// 消えるボールを消す処理
void CDroppingBallBall::EndErasing(int x, int y) {

	// セルが消えるボールのときの処理
	if (StageCell->Get(x, y)&0x40) {

		// セルを空にする
		StageCell->Set(x, y, ' ');
	}
}

// ボールの描画処理
void CDroppingBallBall::Draw() {

	// 画面の解像度を取得する
	float
		sw=Game->GetGraphics()->GetWidth()/MAX_X,
		sh=Game->GetGraphics()->GetHeight()/MAX_Y;

	// ...(中略)...

	// ステージのすべてのセルについて処理する
	for (
		int y=DROPPING_BALL_FIELD_TOP;
		y<DROPPING_BALL_FIELD_BOTTOM;
		y++
	) {
		for (int x=FieldLeft; x<FieldRight; x++) {

			// セルの種類を取得する
```

```
                char c=StageCell->Get(x, y);

                //...(中略)...

                //消えるボール
                if (c&0x40) {

                    //タイマーを使って描画色を決める
                    float f=Time*0.1f;

                    //時間とともにボールがだんだん薄くなるように表示する
                    Game->Texture[TEX_BALL0+((c&0x3f)-'0')]->Draw(
                        x*sw, y*sh, sw, sh,
                        0, 0, 1, 1, D3DXCOLOR(f, f, f, 1)
                    );
                }
            }
        }
    }
```

相手側にボールを降らせる

ボールを消したときに、相手側のフィールドに攻撃ボールを降らせるアクションです。攻撃ボールの個数は、ボールを消した数と連鎖数に応じて変わります。対戦ゲームでは、大量の攻撃ボールを相手に送り込んで、相手のフィールドを埋め尽くすことが目的です。

ボールを消すと、相手側の攻撃ボール数が増加します。相手がボールを着地させると、ステージ上方から攻撃ボールが降ってきます（Fig. 2-87）。ここでは攻撃ボールを透明なボールで表しました。

Fig. 2-87 攻撃ボールが降る

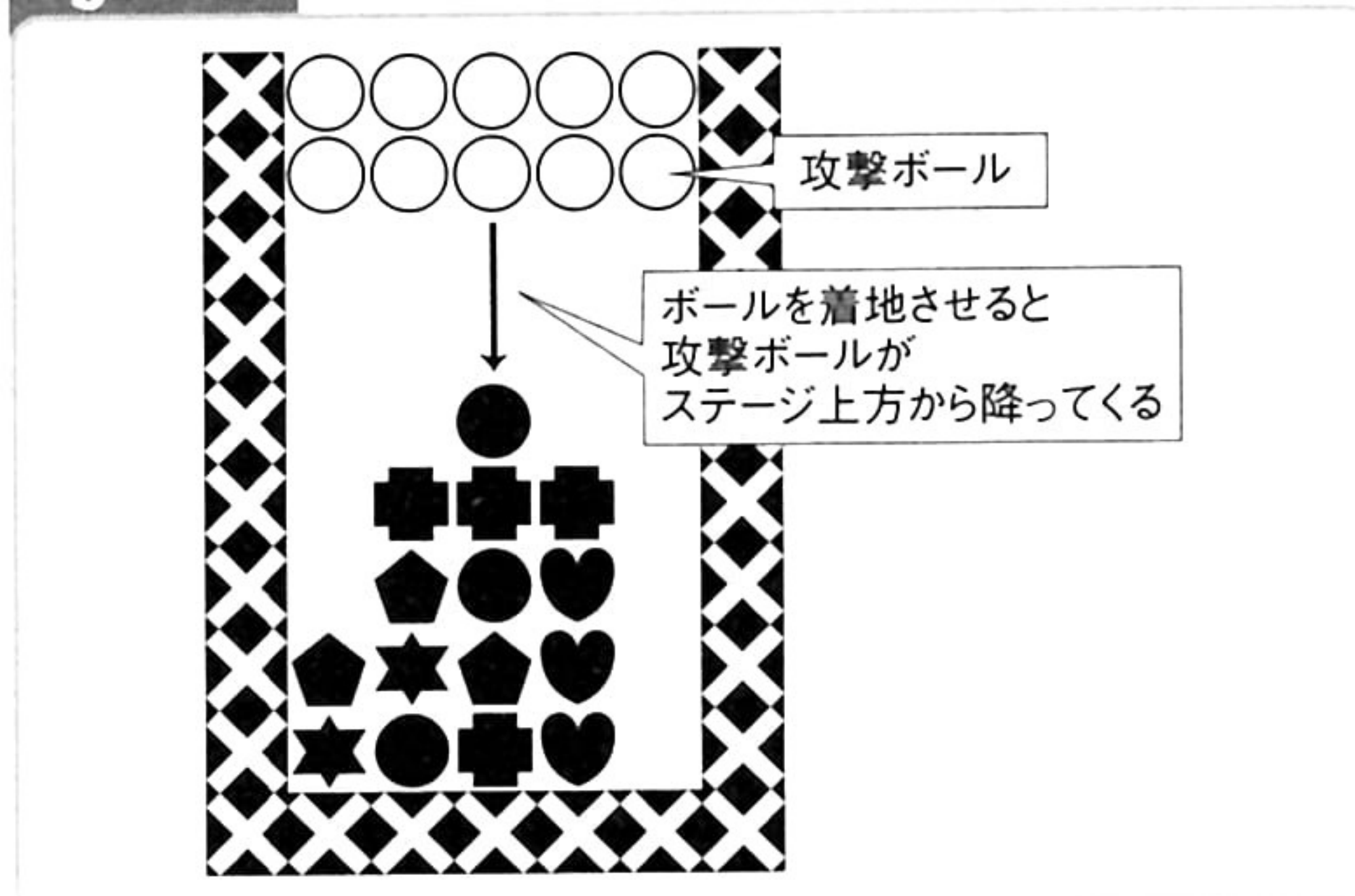
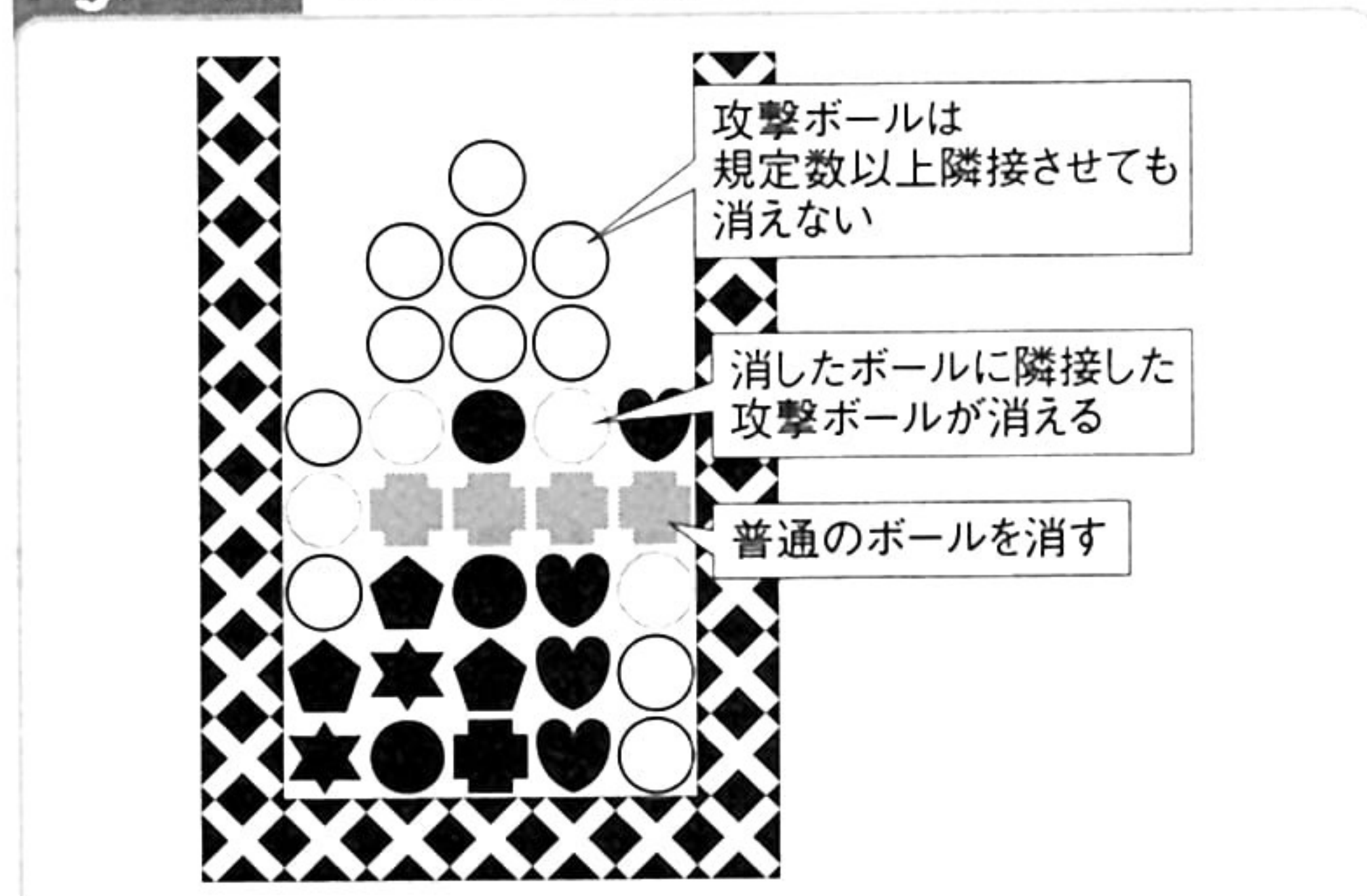



Fig. 2-88 攻撃ボールを消す

攻撃ボールは普通のボールとは違い、隣接させても消すことができません。攻撃ボールを消すには、攻撃ボールに隣接した普通のボールを消します（Fig. 2-88）。

普通のボールを消したときだけではなく、攻撃ボールを消したときにも、相手側の攻撃ボール数が増加します。相手から送り込まれた攻撃ボールを上手に利用すれば、より大きな攻撃を相手に返すこともできます。

相手側にボールを降らせるアクションは『ぷよぷよ』などに採用されています。単にボールを消して生き残るだけではなく、積極的に連鎖消しを狙うことによって、相手に強力な攻撃を送ることができます。相手よりも早く、できるだけ大きな連鎖を組む、という積極的なプレイが、このゲームの魅力です。

アルゴリズム

攻撃ボールを降らせるには、プレイヤーごとに攻撃ボール数を記録しておきます。そして、ボールを消したときに、相手側の攻撃ボール数を増やします。

攻撃ボールの個数は、ボールを消した数と、連鎖消しの回数によって決まります。ボールを同じ4個消した場合でも、1連鎖目と2連鎖目では、相手に送る攻撃ボールの数が変わります。連鎖数が多いほど、攻撃ボール数は爆発的に増えます。

攻撃ボールの数

攻撃ボール数の計算方法は、ゲームによってさまざまです。計算方法を変えると、ゲームバランスが変わります。本書のサンプルでは、次のように攻撃ボール数を計算します。

攻撃ボール数=（消した個数-規定数+１）×連鎖数

消した個数はボールを消した個数です。規定数はボールを消すために必要な最小の隣接数で、ここでは4個です。連鎖数は連鎖消しの回数で、1回から始まり、連鎖しているかぎり2回・3回と増えていきます。

例えば、1連鎖目にボールを4個消したときには、

(4-4+1)×1=1

なので、1個の攻撃ボールが発生します。同じように、2連鎖目にボールを5個消したときには、

(5-4+1)×2=4

なので、4個の攻撃ボールが発生します。また、3連鎖目にボールを6個消したときには、

(6-4+1)×3=9

なので、9個の攻撃ボールが発生します。

連鎖の各段階で発生した攻撃ボールは、攻撃ボール数に加算します。1連鎖目に4個、2連鎖目に5個、3連鎖目に6個のボールを消した場合、発生する攻撃ボールは1個・4個・9個なので、攻撃ボール数は合計14個となります。

相手から送られた攻撃ボールを消した場合にも、相手への攻撃ボール数が増えます。この場合も計算方法はゲームによってさまざまです。本書のサンプルでは、普通のボールを消した個数に、攻撃ボールを消した個数を合計して、上記の計算式を適用します。

例えば4連鎖目に普通のボールを5個、攻撃ボールを3個消したときには、

(5+3-4+1)×4=20

なので、20個の攻撃ボールが発生します。相手の攻撃ボールを上手に利用すると、強力な反撃をすることができます。

ボールを降らせる

実際に攻撃ボールを降らせるのは、相手のボールが着地したときです。ボールが着地したときに、それまでに蓄積された攻撃ボール数に応じて、ステージの上方に攻撃ボールを出現させます（Fig. 2-89）。ここでは数字の「5」で攻撃ボールのセルを表しました。

攻撃ボールの数が多いときには、1段ずつ出現させます。1段出現させ、その段が1段落ちたら、次の1段を出現させます。逆に攻撃ボールが少ないときには、ステージ上端のランダムな位置に出現させます。こういった細かい出現のルールは、ゲームによって異なります。

『ぷよぷよ』などでは、大量の攻撃ボールがあるときに、攻撃ボールを何回かに分けて発生させます。攻撃ボール数にかかわらず、一度に発生するボールの最大数は30個までです。ボールが着地すると、30個のボールが降りますが、次は新しい普通のボールが出現します。残りの攻撃ボールが降るのは、新しいボールが着地した後です。このルールのおかげで、相手の先制攻撃を受けても、自分も追いかけて連鎖を発動させるなどして、反撃することができます。

Fig. 2-89　攻撃ボールの出現

=	5	5	5	5	5	=
=						
=						
=						=
=						=
=			0			=
=		3	3	3		=
=		4	0	2		=
=	4	1	4	3		=
=	1	0	3	2		=
=	=	=	=	=	=	=

プログラム

List 2-14は相手側にボールを降らせるプログラムです。攻撃ボールを出現させる処理を掲載しました。この他に、「連鎖的に消す」(→p. 106) のList 2-13にも、攻撃ボールに関する処理が含まれています。

攻撃ボールを出現させる処理 (List 2-13のAttack関数) では、攻撃ボール数が0より大きいかぎり、攻撃ボールを出現させます。出現位置はフィールドの上端で、左右の位置はランダムに決めます。

相手に送る攻撃ボールの数を決めるのは、ボールの移動処理 (List 2-13のMove関数) と、隣接したボールを消えるボールとしてマークする処理 (List 2-13のBeginErasing関数) で行います。消したボールの数が多いほど、また連鎖数が多いほど、大量の攻撃ボールを相手に送ることができます。

List 2-14 相手側にボールを降らせる (CDroppingBallBallクラス)

```
// 攻撃ボールを出現させる処理
void CDroppingBallBall::Attack() {

    // フィールドの幅
    int n=FieldRight-FieldLeft;

    // 攻撃ボールの出現位置をランダムに決める
    int r=Rand.Int31()%n;

    // 横1段に攻撃ボールを出現させる
    for (int i=0; i<n; i++) {

        // 攻撃ボールを出現させるX座標
        int x=(i+r)%n+FieldLeft;

        // 攻撃ボール数が0より大きいかぎり、ボールを出現させる
        if (AttackCount>0) {

            // 攻撃ボールをステージのセルに書き込む
            StageCell->Set(x, 0, '0'+DROPPING_BALL_COLOR_COUNT);

            // 攻撃ボール数を減少させる
            AttackCount--;
        } else

        // 攻撃ボール数が0ならば、ボールを出現させない
        {
            // セルを空にする
            StageCell->Set(x, 0, ' ');
        }
    }
}
```

一度では消えないボール

周囲にある普通のボールを消すと、普通のボールに変わる、特殊なボールです。普通のボールに変えれば、同じ種類のボールを規定数以上隣接させることによって、消すことができます。

一度では消えないボールは、普通のボールとは少し異なる姿をしています。ここでは普通のボールを透明にして表すことにしましょう (Fig. 2-90)。ゲームによっては、普通のボールと同じ色の、小さめのボールを使う場合もあります。

一度では消えないボールは、普通のボールとは違い、同じ種類のボールを規定数以上並べても消すことができません。ボールを消すには、攻撃ボールを消すのに似た方法を使います。周囲にある普通のボールを消すと、一度では消えないボールは普通のボールに変化します (Fig. 2-91)。なお、ここではボールを消すための規定数を3個としました。

Fig. 2-90 一度では消えないボール

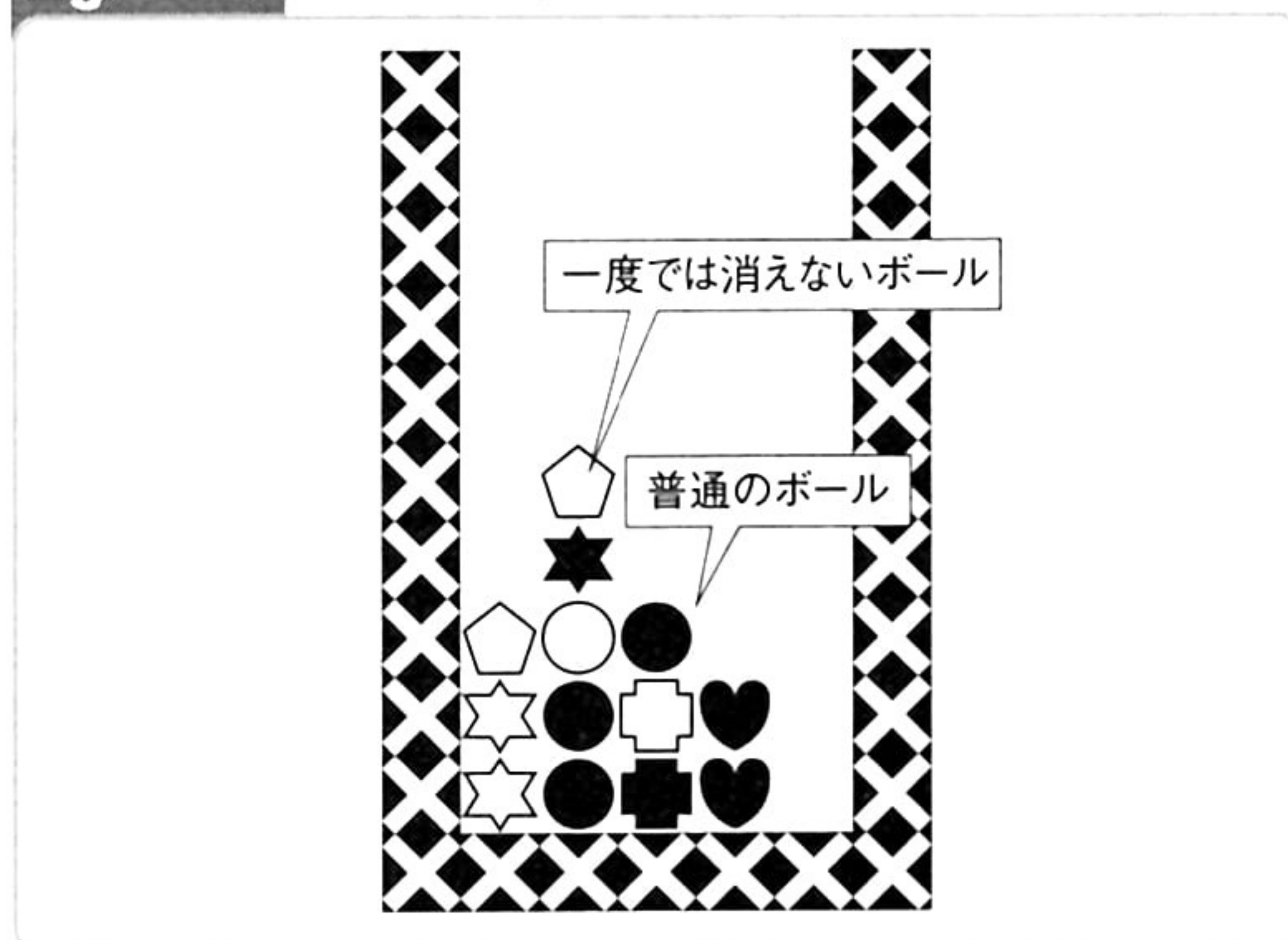



Fig. 2-91 普通のボールに変化する

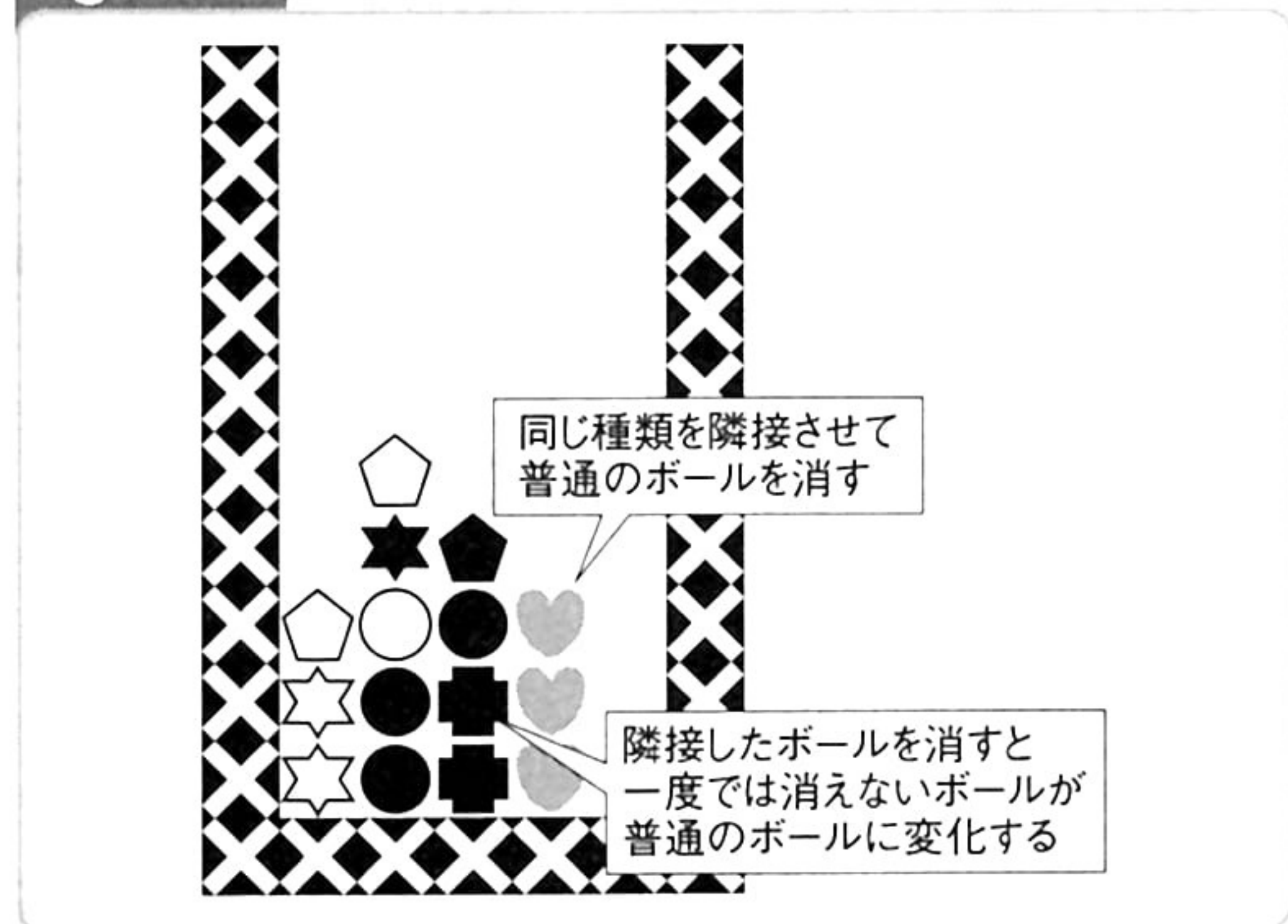



Fig. 2-92 変化したボールが規定数以上隣接すると消える

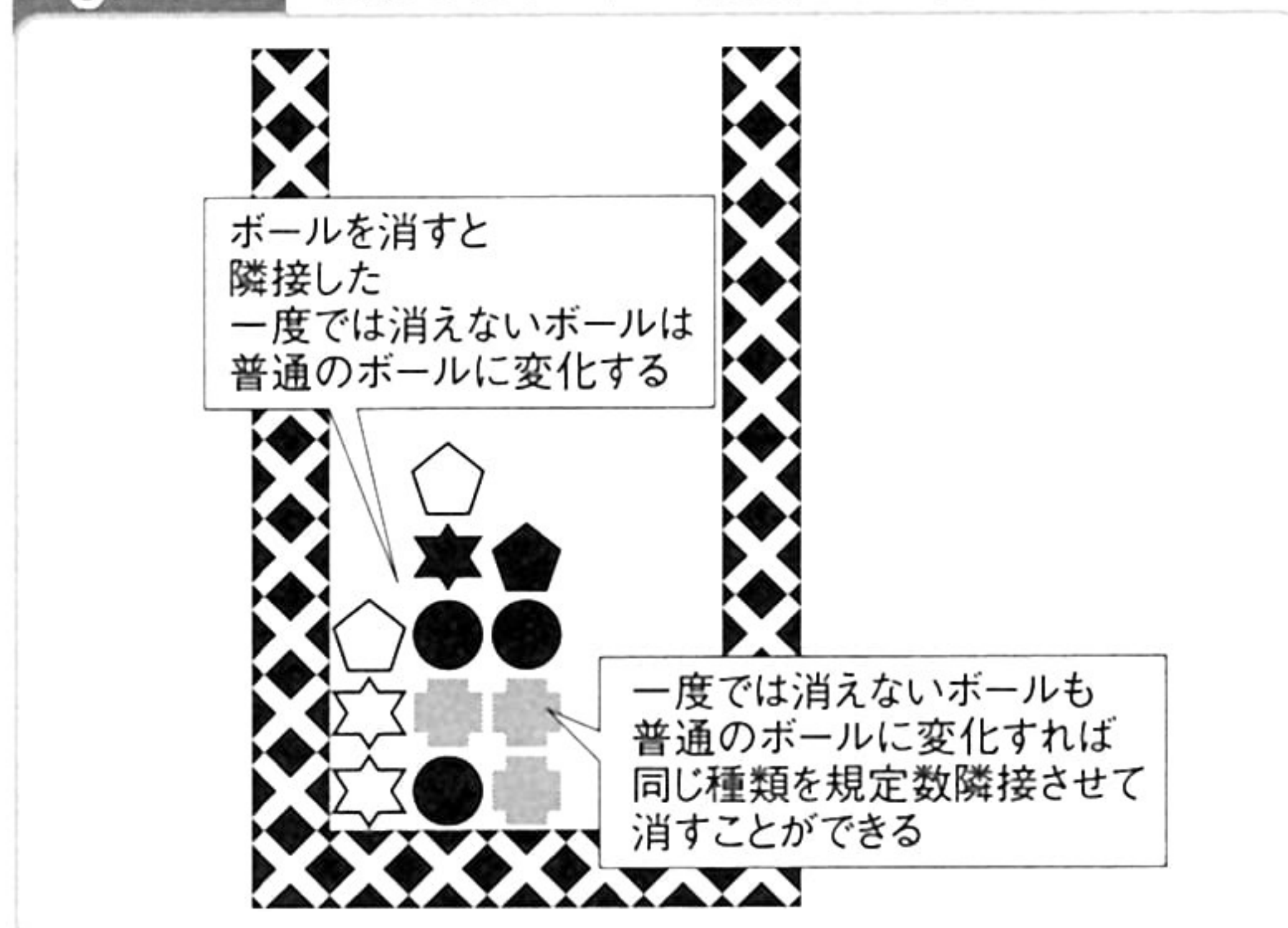
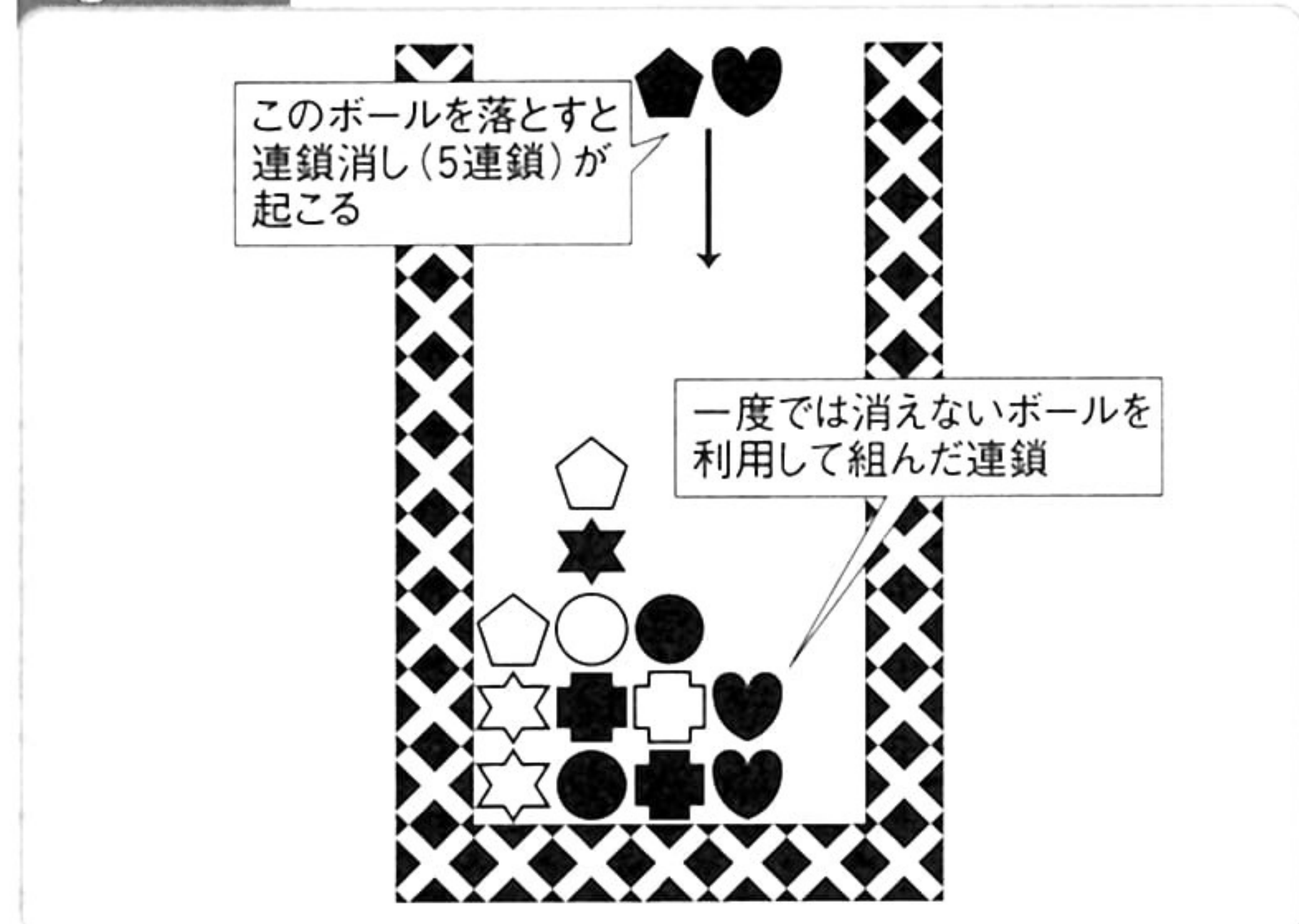



Fig. 2-93 一度では消えないボールを利用した連鎖

普通のボールに変化すれば、あとは同じ種類のボールを規定数以上隣接させれば、消すことができます（Fig. 2-92）。もしも他の一度では消えないボールが周囲にあれば、それらのボールは普通のボールに変化します。この性質を利用して、連鎖を組むことも可能です（Fig. 2-93）。

一度では消えないボールは『対戦ぱずるだま』などに採用されています。このゲームでは、普通のボールに混じって、小さなボールが降ってきます。小さなボールは同じ種類を隣接させても消えませんが、周囲にある普通のボールを消すと、普通のボールに変化します。小さなボールを上手に利用すると、『ぷよぷよ』とは違った方法で連鎖を組むことができます。

アルゴリズム

一度では消えないボールの処理は、「相手側にボールを降らせる」（→p. 115）で解説した、攻撃ボールを消す処理に似ています。周囲にある普通のボールを消すと、一度では消えないボールは普通のボールに変化します。

普通のボールを消すときには、周囲に一度では消えないボールがないかどうかを調べます。

Fig. 2-94　消えるボールとしてマークする

=						=
=						=
=						=
=						=
=						=
=		9				=
=		1	4			=
=	9	5	0	2		=
=	6	3	8	2		=
=	6	0	3	2		=
=	=	=	=	=	=	=

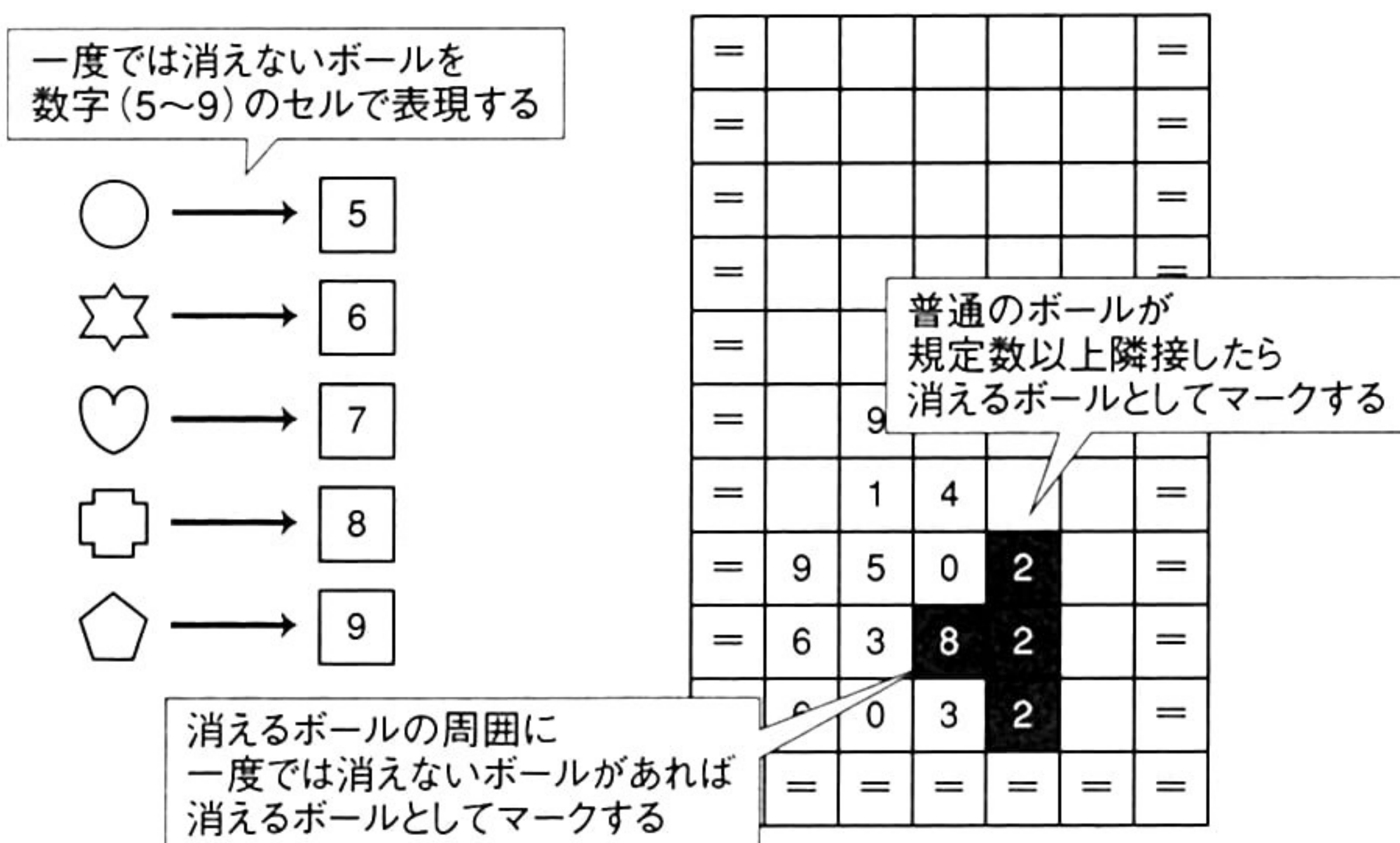
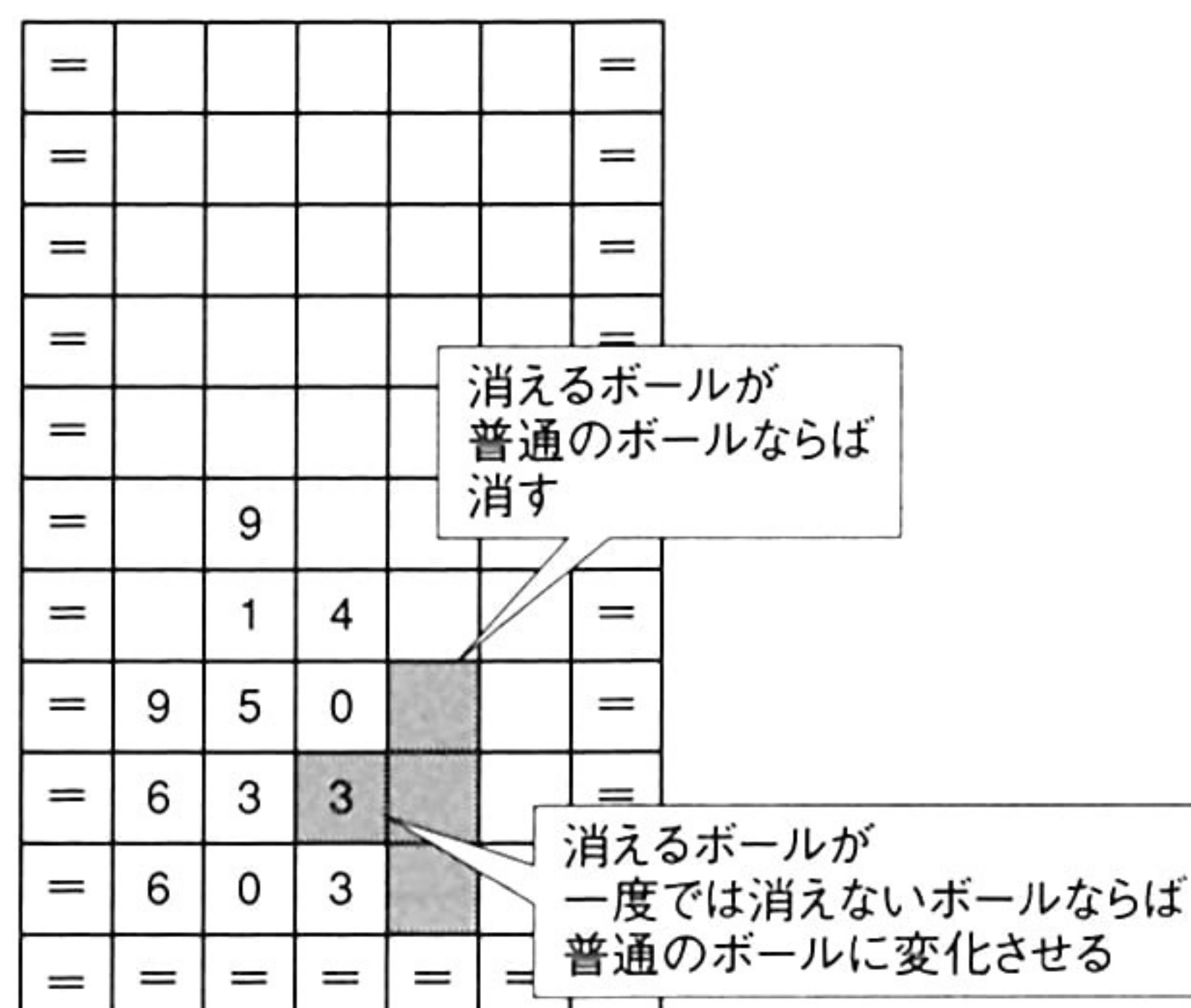



Fig. 2-95　普通のボールに変化させる

=						=
=						=
=						=
=						=
=						=
=		9				=
=		1	4			=
=	9	5	0			=
=	6	3	3			=
=	6	0	3			=
=	=	=	=	=	=	=

ボールがあるならば、消えるボールとしてマークします（Fig. 2-94）。一度では消えないボールのセルは、数字の「5〜9」を使って表しました。

一定時間が経過したら、消えるボールを消します。このとき、消えるボールが普通のボールかどうかを調べます。普通のボールならば消しますが、一度では消えないボールの場合には、普通のボールに変化させます（Fig. 2-95）。

プログラム

List 2-15は一度では消えないボールのプログラムです。隣接したボールを消えるボールとしてマークする処理（BeginErasing関数）と、消えるボールを消す処理（EndErasing関数）を掲載しました。

これらの2つの関数は、「連鎖的に消す」（→p. 106）のプログラム（List 2-13）に含まれる関数を、オーバーライドして一部変更したものです。他の部分はほとんど同じ処理なので、異なる部分だけを仮想関数として切り出しました。

隣接したボールを消えるボールとしてマークする処理（BeginErasing関数）では、一度では消えないボールを見つけたときに、消えるボールとしてマークします。このプログラムでは、セルの上から2番目のビットをセットすることで、マークを付けています。

消えるボールを消す処理（EndErasing関数）では、一度では消えないボールを普通のボールに変化させます。普通のボールはそのまま消します。

List 2-15 一度では消えないボール（CDroppingBall2Ballクラス）

```
// 隣接したボールを消えるボールとしてマークする処理
// 再帰的に呼び出す
void CDroppingBall2Ball::BeginErasing(int x, int y, char ball) {

    // セルの種類を取得する
    char c=StageCell->Get(x, y);

    // セルが指定された種類のボールのときの処理
    if (c==ball) {

        // ボールを消えるボールとしてマークする
        StageCell->Set(x, y, c|0x40);

        // 上下左右に隣接するセルについても、
        // 同じ種類のボールを消す処理を再帰的に行う
        BeginErasing(x-1, y, c);
        BeginErasing(x+1, y, c);
        BeginErasing(x, y-1, c);
        BeginErasing(x, y+1, c);
    } else
```

```
        // セルが一度では消えないボールのときの処理
        if (
            '0'+DROPPING_BALL_COLOR_COUNT<=c &&
            c<'0'+DROPPING_BALL_COLOR_COUNT*2
        ) {
            // ボールを消えるボールとしてマークする
            StageCell->Set(x, y, c|0x40);
        }
    }

    // 消えるボールを消す処理
    void CDroppingBall2Ball::EndErasing(int x, int y) {

        // セルの種類を取得する
        char c=StageCell->Get(x, y);

        // セルが消えるボールのときの処理
        if (c&0x40) {

            // 消えるボールのマークを解除する
            c&=0x3f;

            // 消えるボールが普通のボールのときには、
            // ボールを消す
            if ('0'<=c && c<'0'+DROPPING_BALL_COLOR_COUNT) {
                StageCell->Set(x, y, ' ');
            } else

            // 消えるボールが一度では消えないボールのときには、
            // 普通のボールに変化させる
            {
                StageCell->Set(x, y, c-DROPPING_BALL_COLOR_COUNT);
            }
        }
    }
```

SAMPLE

「DROPPING BALL2」は「一度では消えないボール」「相手側に連鎖しやすいボールを降らせる」のサンプルです。内容は「DROPPING BALL」(→p. 95)に似ています。

レバーの左右(カーソルキーの左右、プレイヤー2はJキーとLキー)でボールが移動し、下(カーソルキーの下、プレイヤー2はKキー)でボールの落下スピードが上がります。ボタン0(Zキー、プレイヤー2は;キー)を押すと、ボールが右回りに回転します。

同じ種類のボールを3個以上隣接させると、ボールを消すことができます。透明なボールは、一度では消えないボールです。周囲にある普通のボールを消すと、普通のボールに変化します。

ボールを消すと、上に載っていたボールが落下します。落下の影響で、再び同じ種類のボールが隣接し、連鎖的に消えることがあります。

左側のフィールドはプレイヤー1、右側のフィールドはプレイヤー2が使います。画面中央に表示されてい

るボールは、次に落ちてくるボールです。左側がプレイヤー1のボール、右側がプレイヤー2のボールです。
ボールを消した数や連鎖数に応じて、相手側のフィールドに攻撃ボールが降ります。攻撃ボールは一度では消えないボールですが、連鎖しやすい配置になっています。攻撃ボールを逆に利用して、相手に反撃を加えることが、このゲームのポイントです。

DROPPING BALL2 → p. 387

3次元のブロックを落とす

立体的なステージで、3次元のブロックを落として積み上げるアクションです。ブロックを移動させたり、回転させたりしながら、隙間なく積み上げることがゲームの目的です。内容は「ブロックを落とす」(→p. 50) に似ていますが、ステージもブロックも立体的なので、操作性や攻略法はかなり違います。

ステージは穴を見下ろしたような形をしています (Fig. 2-96)。立方体を組み合わせた3次元のブロックは、ステージの手前に出現し、時間とともに奥 (Z方向) に向かって落ちていきます。ボタンを押すと、落下スピードを上げることができます。

レバー入力で、ブロックを上下左右(X方向およびY方向)に動かすことができます(Fig. 2-97)。ブロックを上手に動かして、奥から隙間なく詰めていきます。

ブロックを1段隙間なく詰めると、その段を消すことができます。ブロックを消すことができずに、ステージの手前までがブロックに埋まってしまうと、ゲームオーバーです。

3次元のブロックを落とすゲームとしては『ブロックアウト』があります。このゲームは「ブロックを落とす」(→p. 50) で紹介した『テトリス』によく似ていますが、ステージもブロックも3次元になっているため、ゲームの内容はまったく異なります。

このゲームを上手にプレイするには、ステージやブロックの奥行きを正確に把握することが必須です。2次元の画面で3次元の奥行きを把握するのは難しいのですが、このゲームでは奥行きごとにブロックを色分けして、プレイしやすくしています。また、積んであるブロックがよく見えるように、落下中のブロックを透明で表示する工夫も見られます。

Fig. 2-96 穴を見下ろしたような形のステージ

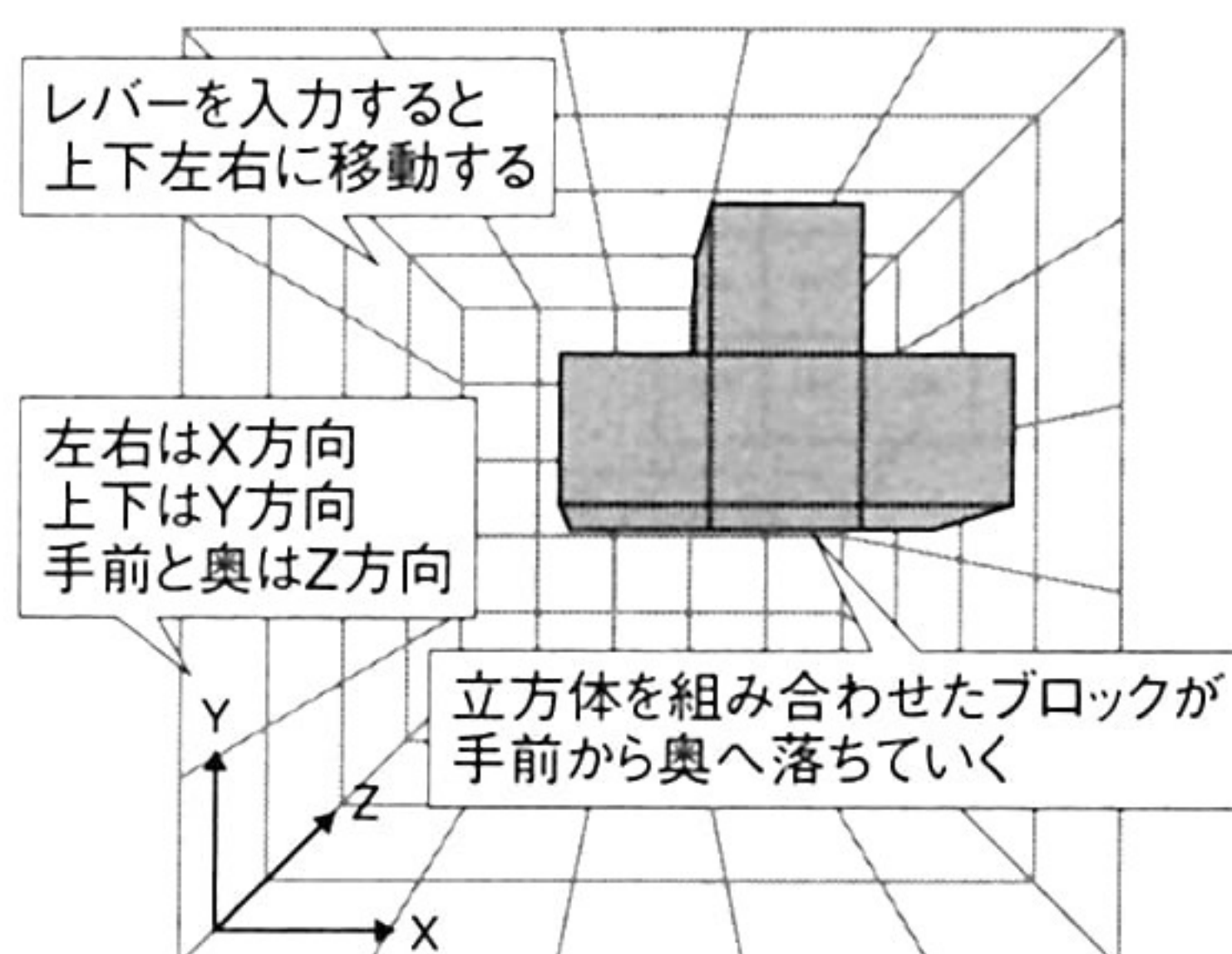
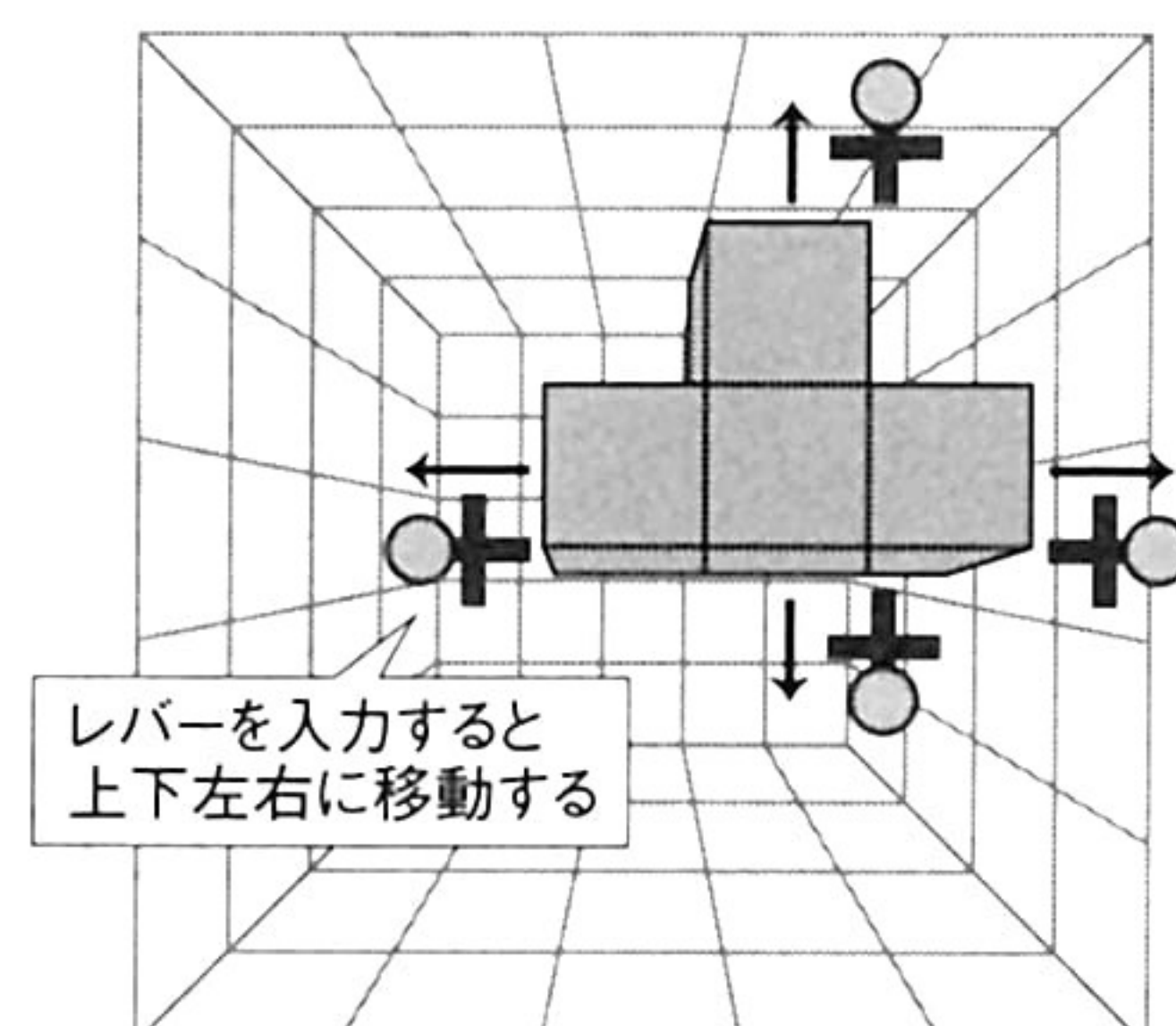



Fig. 2-97 ブロックの移動

本書のサンプルでは、白黒だけで表現する都合上、奥行きによる色分けは行っていません。一方、落下中のブロックは半透明で表示して、積んであるブロックを見やすくしています。

アルゴリズム

3次元のブロックを落とすアクションを実現するには、「ブロックを落とす」(→p. 50) と同様に、ブロックをステージやセルで表現します。立体的なブロックやステージを表すために、3次元のセルを使いますが、考え方は2次元のときと同じです。

まず、ブロックをセルで表現します (Fig. 2-98)。ここでは「3×3×3」のセルを使うことにしましょう。ブロックのセルは文字「#」で表し、空きのセルは空白で表します。

ステージもセルで表現します (Fig. 2-99)。ステージの周囲や底面を囲む壁を、文字「=」で表します。積み上げられたブロックは、文字「#」で表すことにします。

ブロックはタイマーを使って、時間とともに少しずつ落下させます。タイマーが一定値になるたびに、ブロックのセル座標を更新します。ボタンを押したときには、タイマーを待たずにセル座標を更新することによって、ブロックの落下スピードを上げます。

Fig. 2-98 ブロックをセルで表現する

3×3×3のセルでブロックを表現する
(図では3×3のセルを3レイヤー重ねて示した)
奥のレイヤー
中央のレイヤー
手前のレイヤー
ブロックのセル
(#)
空のセル
(空白)

Fig. 2-99 ステージをセルで表現する

3次元のセルでステージを表現する
(図では2次元のセルを複数レイヤー重ねて示した)
一番奥の
レイヤー
奥から2番目の
レイヤー
奥から3番目の
レイヤー
壁および床のセル
(=)
空のセル
(空白)
積み上げられた
ブロックのセル
(#)

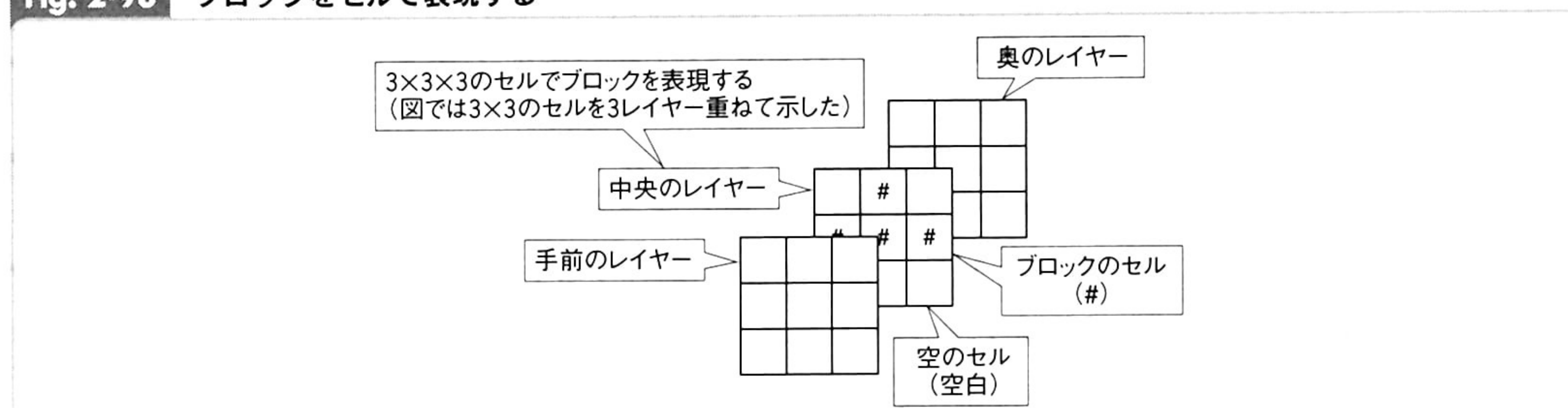
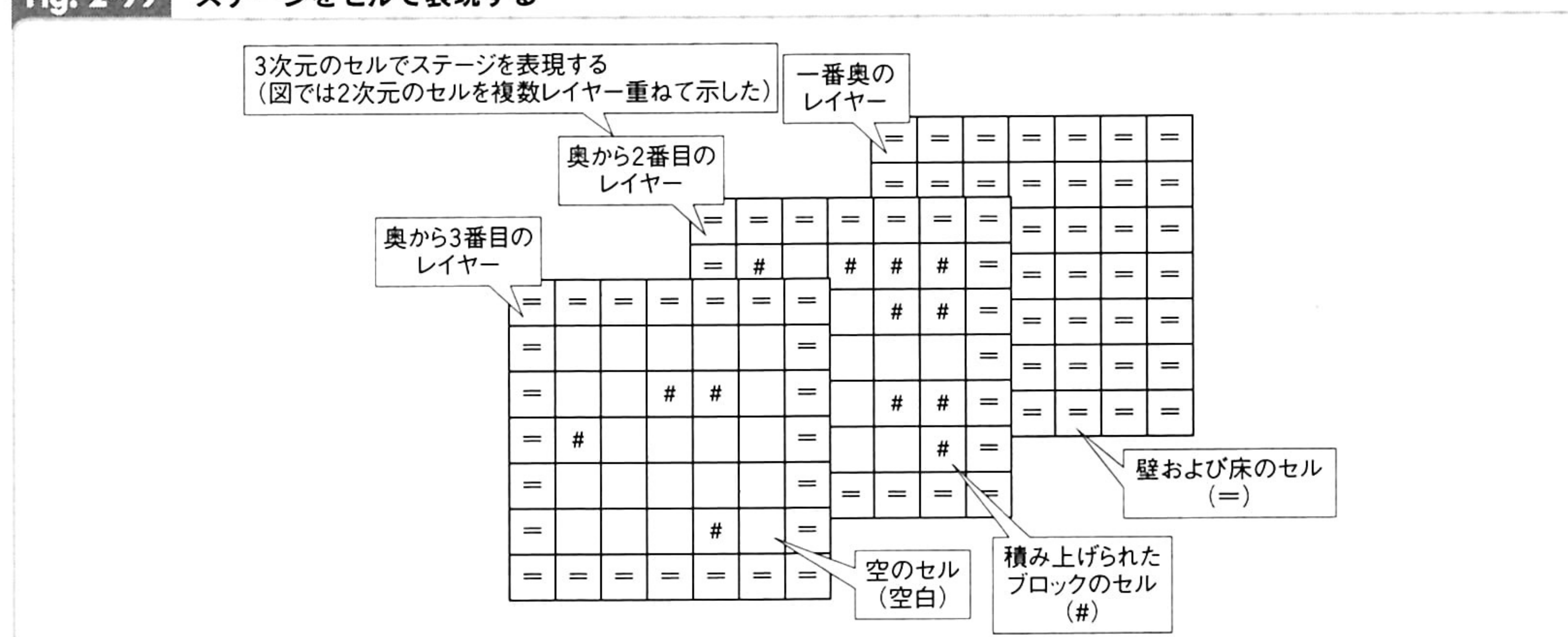



Fig. 2-100 ブロックとステージの当たり判定処理

ステージを
Y方向から
見たところ

=						=
=						=
=		#	#	#		
=						=
=						=
=						
=						
=						=
=			#	#		=
=	#		#	#	#	=
=	=	=	=	=	=	=

ブロックのセル
(3×3×3)

ブロックの空ではないセルと
ステージの空ではないセルとの
接触を調べる

接触しないかぎりは
ブロックを
落下させることができる

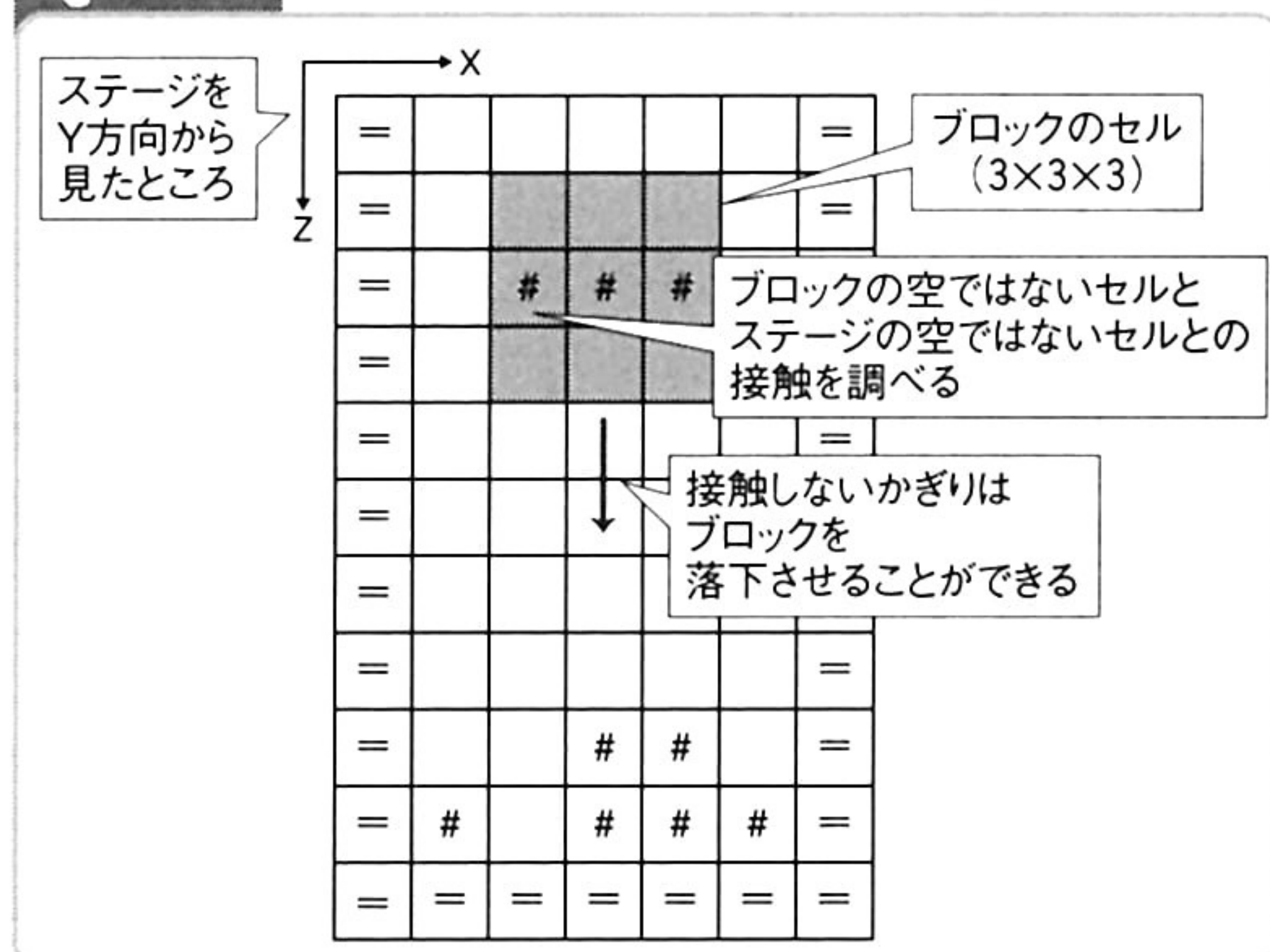
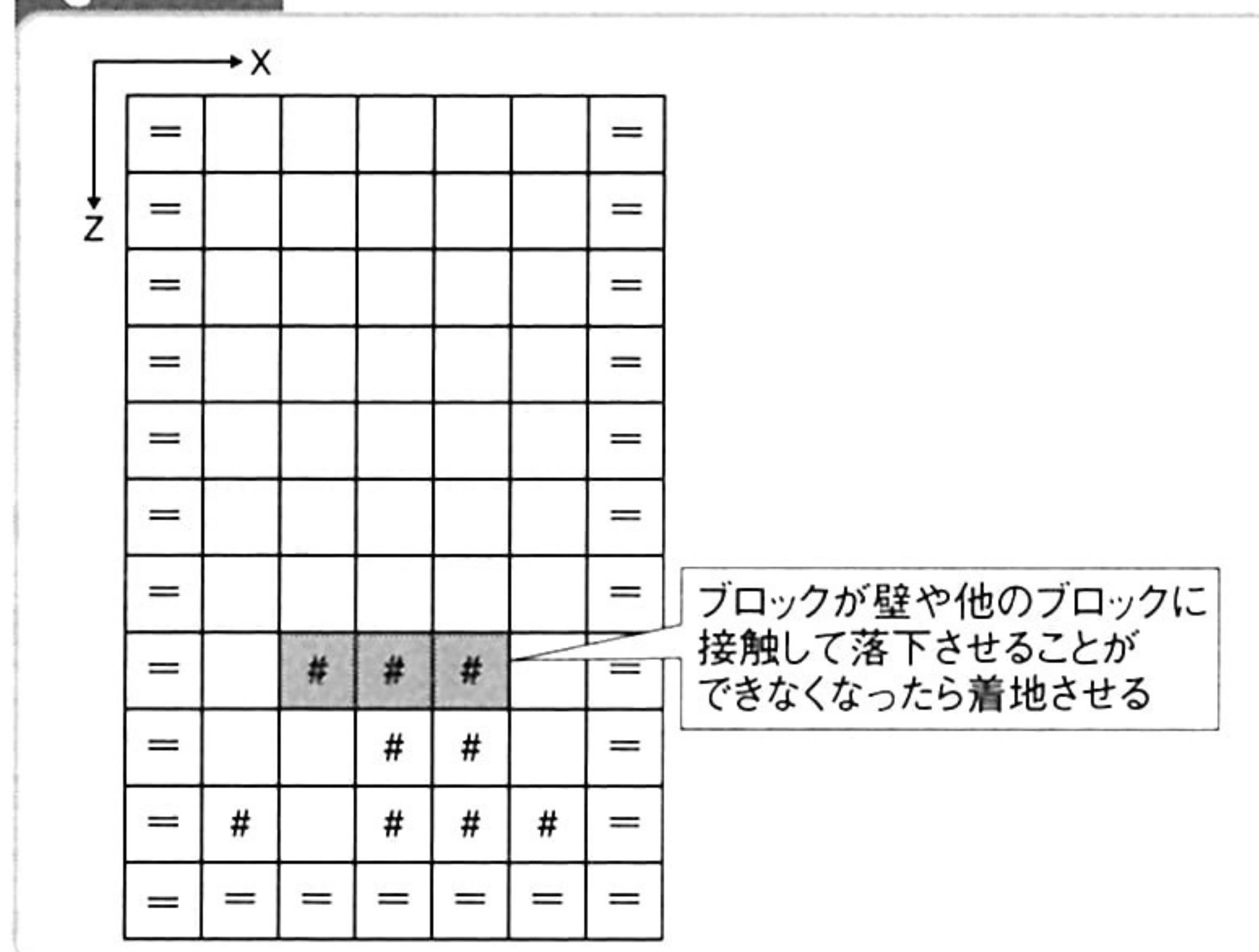



Fig. 2-101 ブロックを着地させる

=						=
=						=
=						=
=						=
=						=
=						=
=						=
=		#	#	#		=
=			#	#		=
=	#		#	#	#	=
=	=	=	=	=	=	=

ブロックが壁や他のブロックに
接触して落下させることが
できなくなったら着地させる

ブロックのセル座標は、(CX, CY, CZ) のような3次元の座標で表します。ブロックを落とすときには、Z座標に「1」を加算して、座標を (CX, CY, CZ+1) にします。

セル座標を更新する際には、ブロックのセルとステージのセルとの間で、当たり判定処理を行います。同じ位置にあるセルの内容を調べて、どちらのセルも空ではないときには、接触したと判定します。ブロックがステージの壁や他のブロックに接触しなければ、落下させることができます (Fig. 2-100)。図はステージを横 (Y方向) から見た様子です。

ブロックがステージの壁や他のブロックに接触するときには、落下させることができません。このときは、その場所にブロックを着地させます (Fig. 2-101)。着地したブロックのセルは、ステージのセルに合成します。

ブロックを移動させる

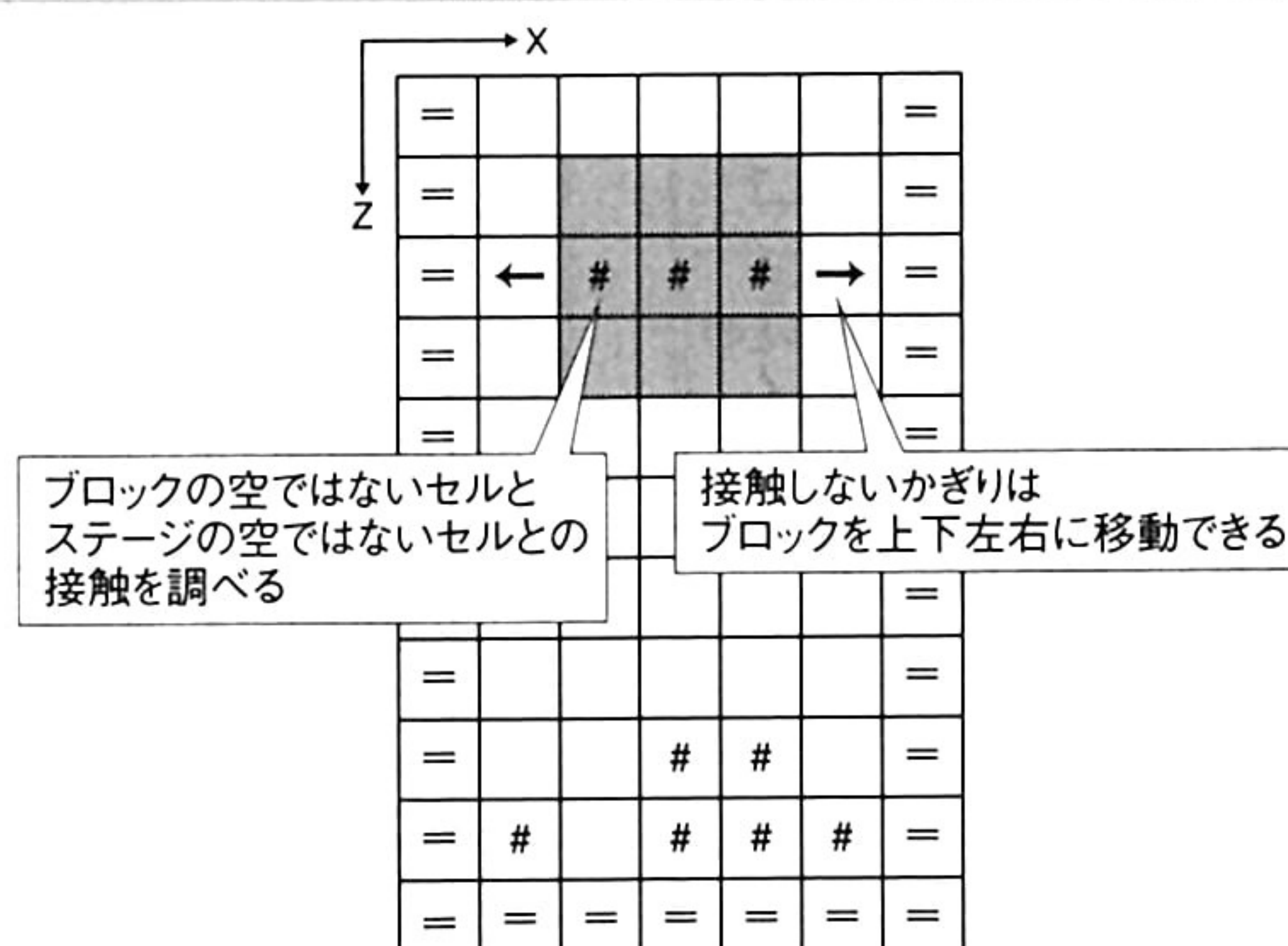
レバーを入力したときには、ブロックを上下左右に移動します。例えばレバーを右に入力したときには、セル座標を (CX, CY, CZ) から (CX+1, CY, CZ) に変化させます。

セル座標を更新する前には、ブロックのセルとステージのセルとの間で当たり判定処理を行います。ブロックがステージの壁や他のブロックに接触しない場合にかぎって、ブロックを移動することができます (Fig. 2-102)。

ブロックを落下させたり、移動したりするときには、セル座標とともに描画座標も更新します。描画座標を (X, Y, Z) とすると、

(X, Y, Z+0.1)
(X, Y, Z+0.2)
(X, Y, Z+0.3)
⋮

のように、少しずつ更新します。これでブロックの落下や移動が滑らかになります。

Fig. 2-102 ブロックの移動

プログラム

List 2-16は3次元のブロックを落とすプログラムです。ブロックの移動処理を掲載しました。

ブロックには入力状態と移動状態があります。入力状態では、時間とともにブロックを落下させます。また、ボタン入力があったときには落下スピードを上げます。レバー入力があったときには、ブロックを上下左右に移動します。

落下・移動のいずれの場合も、ブロックがステージに接触しなければ、セル座標を更新し、移動状態に移行します。移動状態では、タイマーを使って、ブロックの描画座標を少しずつ更新します。これは画面上でブロックを滑らかに動かすためです。

プログラムの内容は「ブロックを落とす」(→p. 50)とほとんど同じですが、セルが3次元になったことが違います。3次元のセルに関する処理の詳細は、CCell3Dクラス(付録CD-ROMの「Puzzle¥Stage.cpp」)を参照してください。

List 2-16 3次元のブロックを落とす(CDroppingBlock3DBlockクラス)

```
// ブロックの移動処理
bool CDroppingBlock3DBlock::Move(const CInputState* is) {

    // 入力状態
    if (State==0) {

        // 落下タイマーの更新
        DropTime++;

        // ボタン3を入力するか、
        // 落下タイマーが一定値に達したら、ブロックを落下させる
        if ((is->Button[3] && !PrevDown)|| DropTime==60) {

            // セル座標を更新したときに、
            // ブロックがステージの壁や他のブロックに 接触するかどうかを調べる
```

```
    if (StageCell->Hit(CX, CY, CZ+1, BlockCell)) {

        //接触する場合には、ブロックのセルをステージのセルに合成する
        StageCell->Merge(CX, CY, CZ, BlockCell);

        //着地状態に移行する
        State=2;
    } else {

        //接触しない場合には、
        //落下タイマーや速度を設定し、セル座標を更新する
        DropTime=0;
        CZ++;
        VX=0;
        VY=0;
        VZ=1;

        // 移動状態に移行する
        Time=0;
        State=1;
    }
} else

//レバーを左に入力しており、
//かつブロックが移動先でステージに接触しなければ、
//ブロックを左に移動させる
if (is->Left && !StageCell->Hit(CX-1, CY, CZ, BlockCell)) {

    //セル座標の更新
    CX--;

    //速度の設定
    VX=-1;
    VY=0;
    VZ=0;

    //移動状態に移行する
    Time=0;
    State=1;
} else

//レバーを右に入力しており、
//かつブロックが移動先でステージに接触しなければ、
//ブロックを右に移動させる
if (is->Right && !StageCell->Hit(CX+1, CY, CZ, BlockCell)) {

    //セル座標の更新
    CX++;
```

```
        // 速度の設定
        VX=1;
        VY=0;
        VZ=0;

        // 移動状態に移行する
        Time=0;
        State=1;
    } else

    // レバーを上に入力しており、
    // かつブロックが移動先でステージに接触しなければ、
    // ブロックを上に移動させる
    if (is->Up && !StageCell->Hit(CX, CY+1, CZ, BlockCell)) {

        // セル座標の更新
        CY++;

        // 速度の設定
        VX=0;
        VY=1;
        VZ=0;

        // 移動状態に移行する
        Time=0;
        State=1;
    } else

    // レバーを下に入力しており、
    // かつブロックが移動先でステージに接触しなければ、
    // ブロックを下に移動させる
    if (is->Down && !StageCell->Hit(CX, CY-1, CZ, BlockCell)) {

        // セル座標の更新
        CY--;

        // 速度の設定
        VX=0;
        VY=-1;
        VZ=0;

        // 移動状態に移行する
        Time=0;
        State=1;
    } else

    // ...(中略)...
}
```

```
    //移動状態の処理
    if (State==1) {

        //タイマーの更新
        Time++;

        //タイマーを使って、描画座標を少しずつ更新する
        X=CX-VX*(1-Time*0.1f);
        Y=CY-VY*(1-Time*0.1f);
        Z=CZ-VZ*(1-Time*0.1f);

        //タイマーが一定値に達したら、入力状態に移行する
        if (Time==10) {
            State=0;
        }
    }

    //…(中略)…
}
```

SAMPLE

「DROPPING BLOCK 3D」は「3次元のブロックを落とす」「3次元のブロックを回転させる」「3次元のブロックを1段揃えて消す」のサンプルです。

レバーの上下左右(カーソルキーの上下左右)でブロックが移動し、ボタン3(Vキー)でブロックの落下スピードが上がります。ボタン0〜2(Zキー、Xキー、Cキー)を押すとブロックが回転します。

ブロックを1段に隙間なく詰めると、その段を消すことができます。ブロックがステージの手前まで積もってしまうと、ゲームオーバーです。

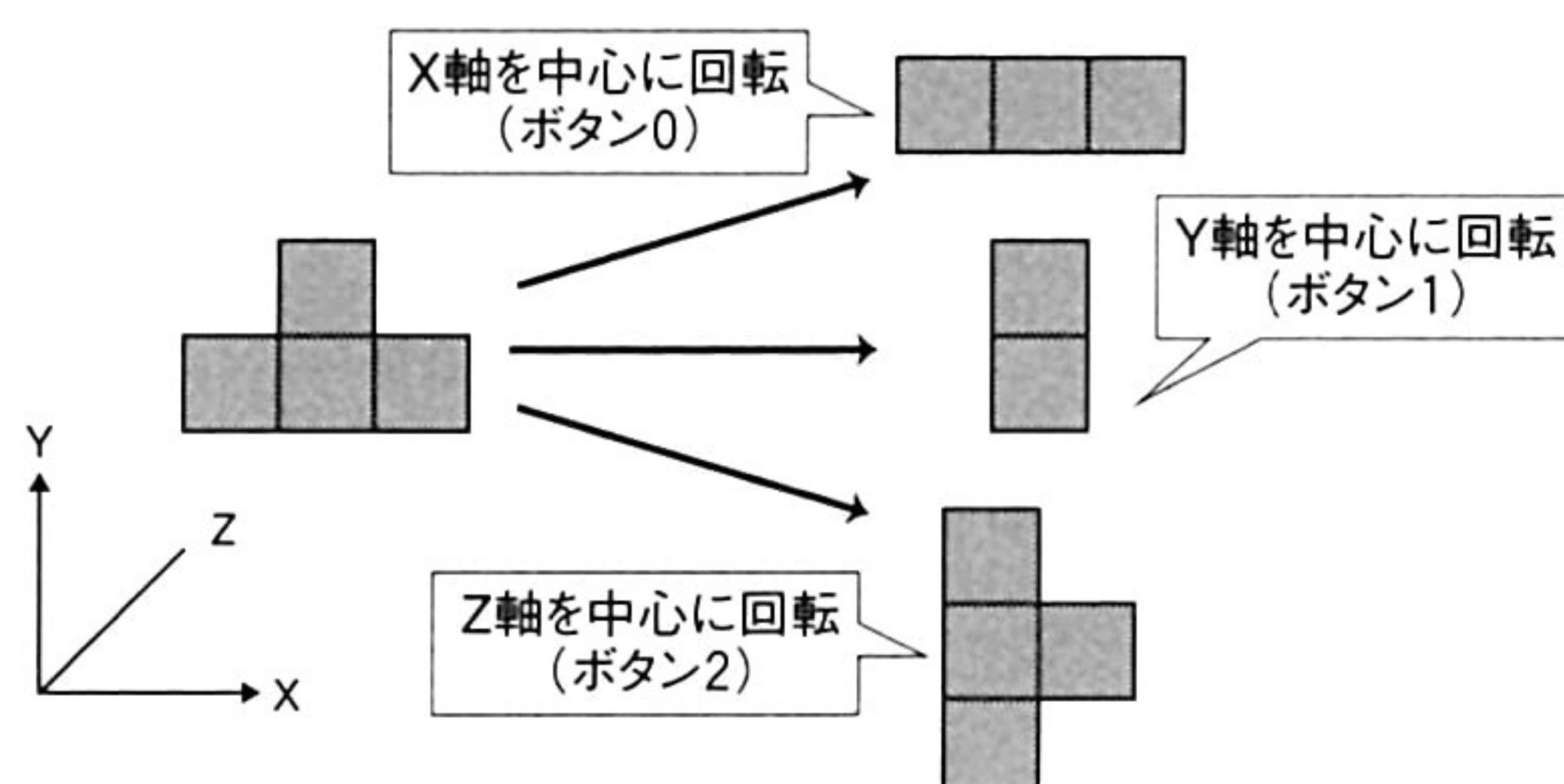
DROPPING BLOCK 3D → p. 387

3次元のブロックを回転させる

ボタン入力で3次元のブロックを回転させるアクションです。移動と回転を組み合わせることによって、ブロックを自由自在に動かし、隙間なく積み上げることができます。

ボタンを1回押すたびに、ブロックは90度ずつ回転します。2次元の場合とは違い、3次元の場合には回転軸が3通りあるので、3種類のボタンを使います(Fig. 2-103)。

2次元の場合と同じく、ブロックにはいろいろな形があります。どのブロックも、ボタンを1回押すたびに、対応する回転軸に対して90度ずつ回転します。

Fig. 2-103 ブロックの回転

アルゴリズム

3次元のブロックを回転させるには、ボタン入力に応じて、セルを移動します。回転軸を中心として、全体が90度回転するようにセルを移動すれば、回転後の状態を作り出すことができます（Fig. 2-104）。

ブロックに3×3×3のセルを使う場合には、回転軸となる3個のセルは動かず、残った24個のセルが動きます。回転軸は3通りあるので、セルを入れ替える処理も3通り用意する必要があります。

ブロックを回転させるときには、回転ができるかどうかを判定するために、ブロックのセルとステージのセルとの間で当たり判定処理を行います（Fig. 2-105）。接触する場合には、ブロックを回転させません。

回転後のブロックがステージに接触するときに、ブロックを上下左右に移動させて、障害物を回避する方法もあります。これは「障害物を避けながらボールを回転させる」（→p. 95）と同様の処理です。混み合ったステージでもブロックが回転させやすくなり、操作性が向上します。

Fig. 2-104 セルを移動してブロックを回転させる

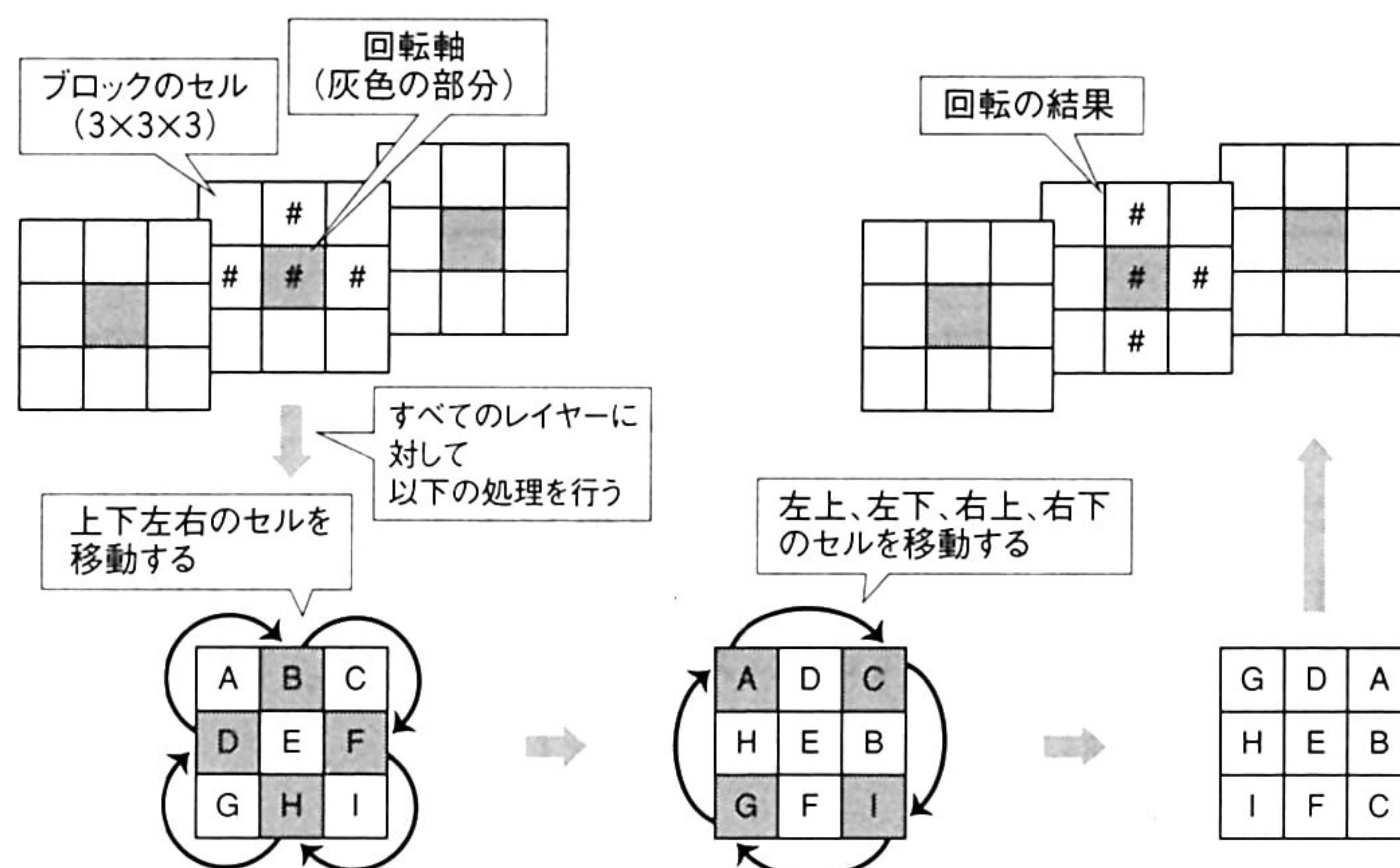
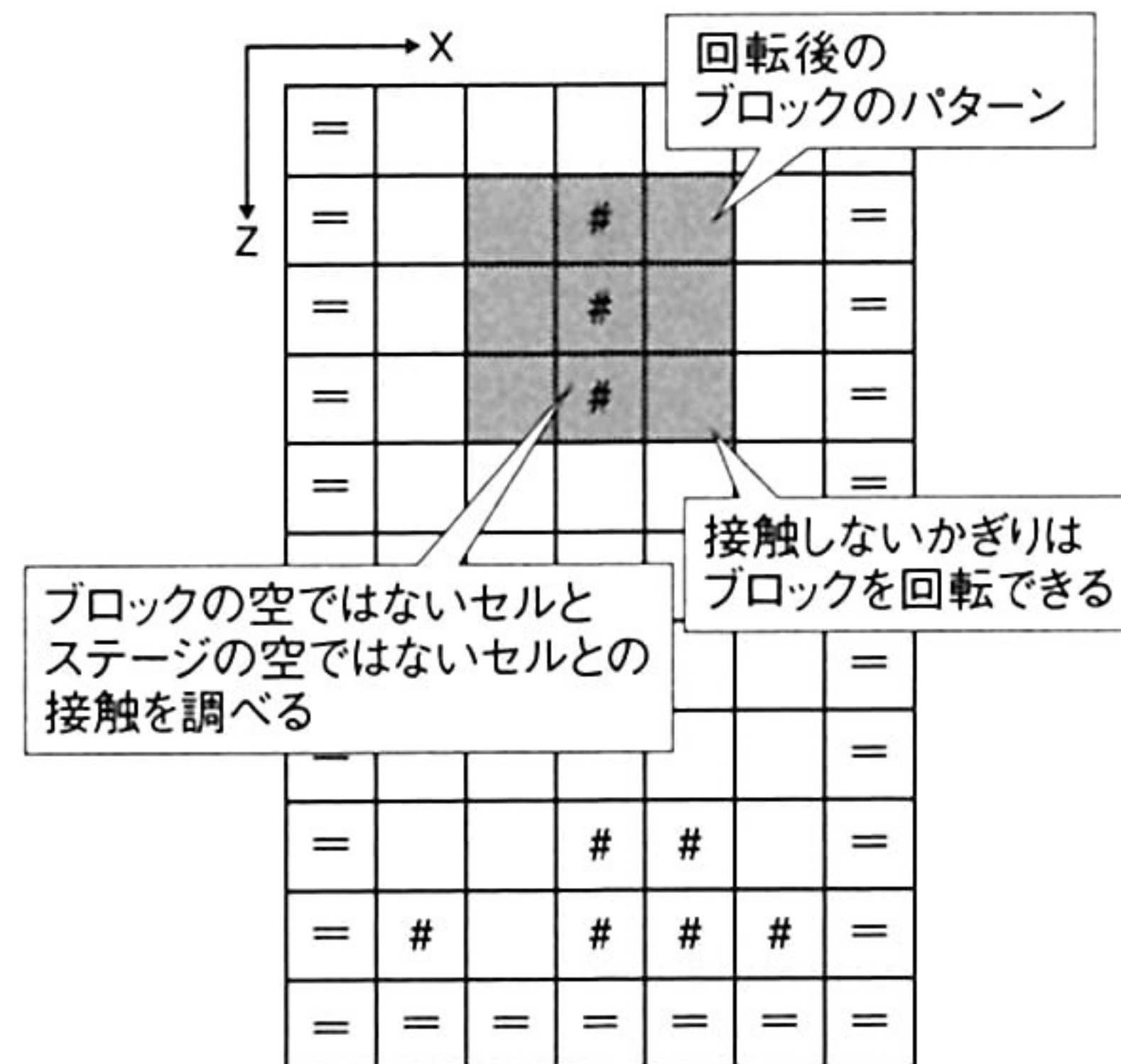



Fig. 2-105 回転できるかどうかの判定処理

回転後の
ブロックのパターン

接触しないかぎりは
ブロックを回転できる

ブロックの空ではないセルと
ステージの空ではないセルとの
接触を調べる

プログラム

List 2-17は3次元のブロックを回転させるプログラムです。ブロックのパターンをデータ化した配列と、ブロックの移動処理を掲載しました。

ブロックのパターンは文字列で表現しました。ここでは簡単にするため、3×3×3のセルのうち、中央にある3×3の平面に配置するブロックだけをデータにしています。両端の3×3の平面にもブロックを配置することはできますが、その場合は3×3×3のデータが必要です。

ブロックは7種類のパターンを用意しました。内容は2次元の「ブロックを回転させる」(→p. 60)に似せています。パターンの内容は自由に変更してもかまいません。

ブロックの移動処理では、ボタンの入力を検出し、ブロックを回転させます。回転軸は押したボタンによって変わります。

回転後のブロックがステージに接触するときには、ブロックの回転を取り消します。ブロックの回転と取り消しは、セルの入れ替えによって行います。セルを入れ替える順番を逆にすると、回転を取り消すことができます。

List 2-17 3次元のブロックを回転させる(CDroppingBlock3DBlockクラス)

```
// ブロックのパターン
// ここでは3×3×3のセルのうち、
// 中央にある3×3の平面に配置するブロックだけをデータにしている
char* DroppingBlock3DPattern[
    DROPPING_BLOCK3D_PATTERN_COUNT]={
    "## "
    "## "
    "   "
    ,
    "   "
```

```
        "###"
        "   "
        ,
        "## "
        " ##"
        "   "
        ,
        " ##"
        "## "
        "   "
        ,
        " # "
        "###"
        "   "
        ,
        "#  "
        "###"
        "   "
        ,
        "  #"
        "###"
        "   "
};

// ブロックの移動処理
bool CDroppingBlock3DBlock::Move(const CInputState* is) {

    // 入力状態の処理
    if (State==0) {

        // ...(中略)...

        // 直前にボタンを離しているときだけ処理する
        if (!PrevButton) {

            // ボタン0を入力したときには、
            // X方向を回転軸としてブロックを回転させる
            if (is->Button[0]) {
                for (int x=0; x<DROPPING_BLOCK3D_CELL_SIZE; x++) {
                    BlockCell->Swap(x, 1, 2, x, 2, 1);
                    BlockCell->Swap(x, 0, 1, x, 1, 2);
                    BlockCell->Swap(x, 1, 0, x, 0, 1);
                    BlockCell->Swap(x, 0, 0, x, 0, 2);
                    BlockCell->Swap(x, 2, 0, x, 0, 0);
                    BlockCell->Swap(x, 2, 2, x, 2, 0);
                }

                // 回転後のブロックとステージが接触するときには、
                // 回転を取り消す
                if (StageCell->Hit(CX, CY, CZ, BlockCell)) {
```

```
            for (int x=0; x<DROPPING_BLOCK3D_CELL_SIZE; x++) {
                BlockCell->Swap(x, 2, 2, x, 2, 0);
                BlockCell->Swap(x, 2, 0, x, 0, 0);
                BlockCell->Swap(x, 0, 0, x, 0, 2);
                BlockCell->Swap(x, 1, 0, x, 0, 1);
                BlockCell->Swap(x, 0, 1, x, 1, 2);
                BlockCell->Swap(x, 1, 2, x, 2, 1);
            }
        }
    } else

    // ボタン1を入力したときには、
    // Y方向を回転軸としてブロックを回転させる
    if (is->Button[1]) {
        for (int y=0; y<DROPPING_BLOCK3D_CELL_SIZE; y++) {
            BlockCell->Swap(2, y, 1, 1, y, 2);
            BlockCell->Swap(1, y, 0, 2, y, 1);
            BlockCell->Swap(0, y, 1, 1, y, 0);
            BlockCell->Swap(0, y, 0, 0, y, 2);
            BlockCell->Swap(2, y, 0, 0, y, 0);
            BlockCell->Swap(2, y, 2, 2, y, 0);
        }

        // 回転後のブロックとステージが接触するときには、
        // 回転を取り消す
        if (StageCell->Hit(CX, CY, CZ, BlockCell)) {
            for (int y=0; y<DROPPING_BLOCK3D_CELL_SIZE; y++) {
                BlockCell->Swap(2, y, 2, 2, y, 0);
                BlockCell->Swap(2, y, 0, 0, y, 0);
                BlockCell->Swap(0, y, 0, 0, y, 2);
                BlockCell->Swap(0, y, 1, 1, y, 0);
                BlockCell->Swap(1, y, 0, 2, y, 1);
                BlockCell->Swap(2, y, 1, 1, y, 2);
            }
        }
    } else

    // ボタン2を入力したときには、
    // Z方向を回転軸としてブロックを回転させる
    if (is->Button[2]) {
        for (int z=0; z<DROPPING_BLOCK3D_CELL_SIZE; z++) {
            BlockCell->Swap(1, 2, z, 2, 1, z);
            BlockCell->Swap(0, 1, z, 1, 2, z);
            BlockCell->Swap(1, 0, z, 0, 1, z);
            BlockCell->Swap(0, 0, z, 0, 2, z);
            BlockCell->Swap(2, 0, z, 0, 0, z);
            BlockCell->Swap(2, 2, z, 2, 0, z);
        }
        // 回転後のブロックとステージが接触するときには、
        // 回転を取り消す
```

```
                if (StageCell->Hit(CX, CY, CZ, BlockCell)) {
                    for (int z=0; z<DROPPING_BLOCK3D_CELL_SIZE; z++) {
                        BlockCell->Swap(2, 2, z, 2, 0, z);
                        BlockCell->Swap(2, 0, z, 0, 0, z);
                        BlockCell->Swap(0, 0, z, 0, 2, z);
                        BlockCell->Swap(1, 0, z, 0, 1, z);
                        BlockCell->Swap(0, 1, z, 1, 2, z);
                        BlockCell->Swap(1, 2, z, 2, 1, z);
                    }
                }
            }
        }

        //直前にボタンを押したかどうかを記録しておく
        PrevButton=is->Button[0]|is->Button[1]|is->Button[2];

        //ボタンを押し続けたときに、ブロックが次々に落ちないようにするための処理
        if (!is->Button[3]) PrevDown=false;
    }

    //...(中略)...
}
```

3次元のブロックを1段揃えて消す

ブロックを同じ奥行きで1段揃えると、揃えた段のブロックが消えるというアクションです。移動や回転を駆使して、ブロックを隙間なく積み上げ、次々に消していくことがゲームの目的です。効率よく消さないと、すぐにステージがブロックで埋まってしまいます。

ブロックを消すには、同じ奥行きの段に、ブロックを隙間なく詰めます（Fig. 2-106）。ブロックが1段揃うと、その段は消えます。

Fig. 2-106 ブロックを隙間なく詰めて消す

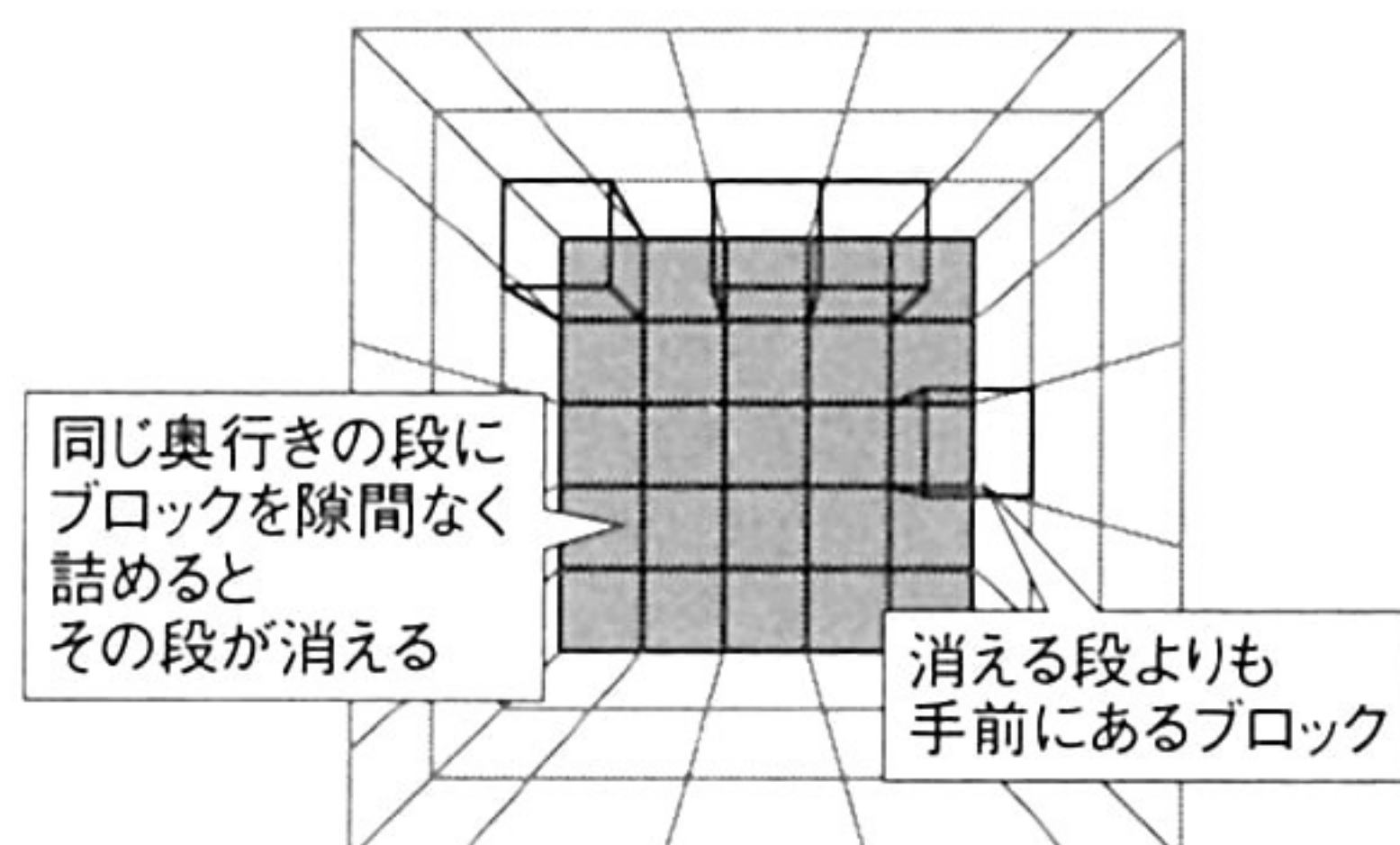
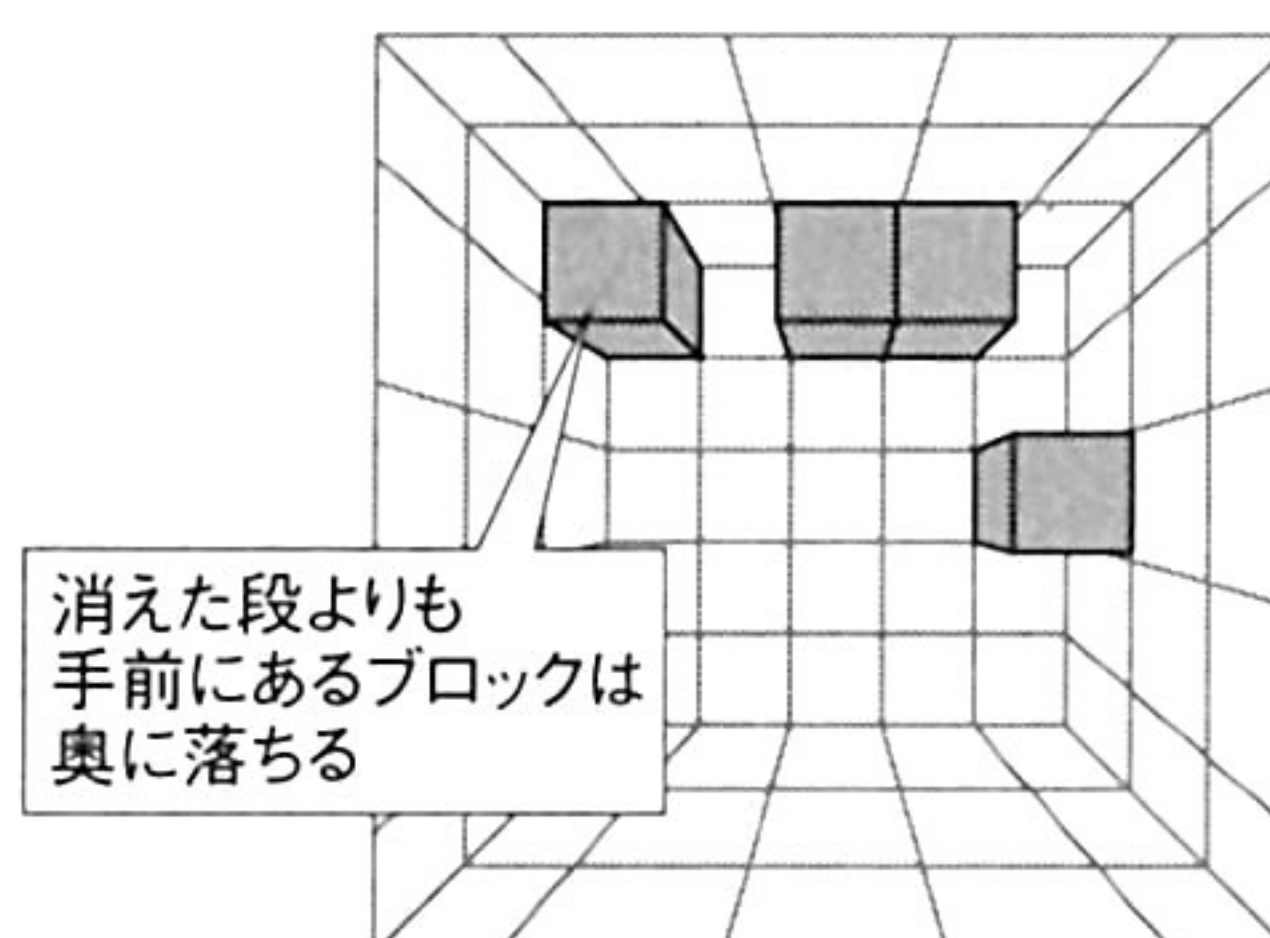



Fig. 2-107 手前にあるブロックが落ちていく

ブロックが消えると、消えた段よりも手前にあるブロックが落ちていきます（Fig. 2-107）。上手にブロックを積むと、1段とはかぎらず、複数の段をまとめて消すこともできます。

アルゴリズム

ブロックを消すには、ブロックが着地したときに、ステージ全体のセルを調べます。ブロックが隙間なく揃っている段、つまり空のセルが1個もない段があったら、その段は消すことができます（Fig. 2-108）。

消える段が見つかったら、その段にマークを付けます。ここでは、ブロックのセルを表す文字「#」を、文字「+」に変更して、消えるブロックを表すことにします（Fig. 2-109）。

一定時間が経過したら、消えるブロックの段を完全に消します。そして、消えた段よりも手前にあるすべての段を、奥行き方向に落とします（Fig. 2-110）。

消えるブロックをマークするのは、ブロックが消えるアニメーションを表示するためです。サンプルでは、消えるブロックがしだいに薄くなって消えるような演出を行います。

Fig. 2-108 消える段を探す

空のセルがまったくない段は
消すことができる

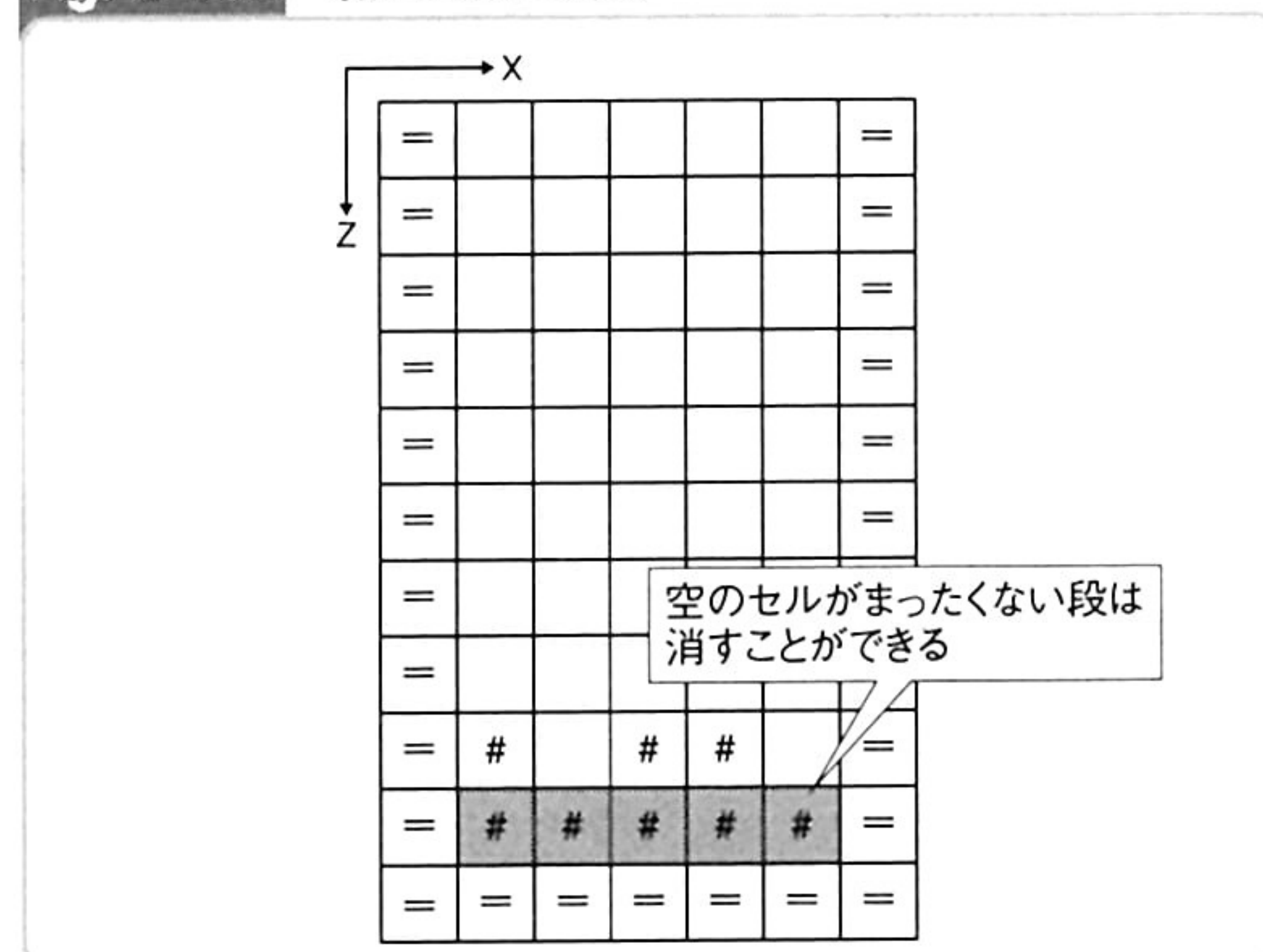
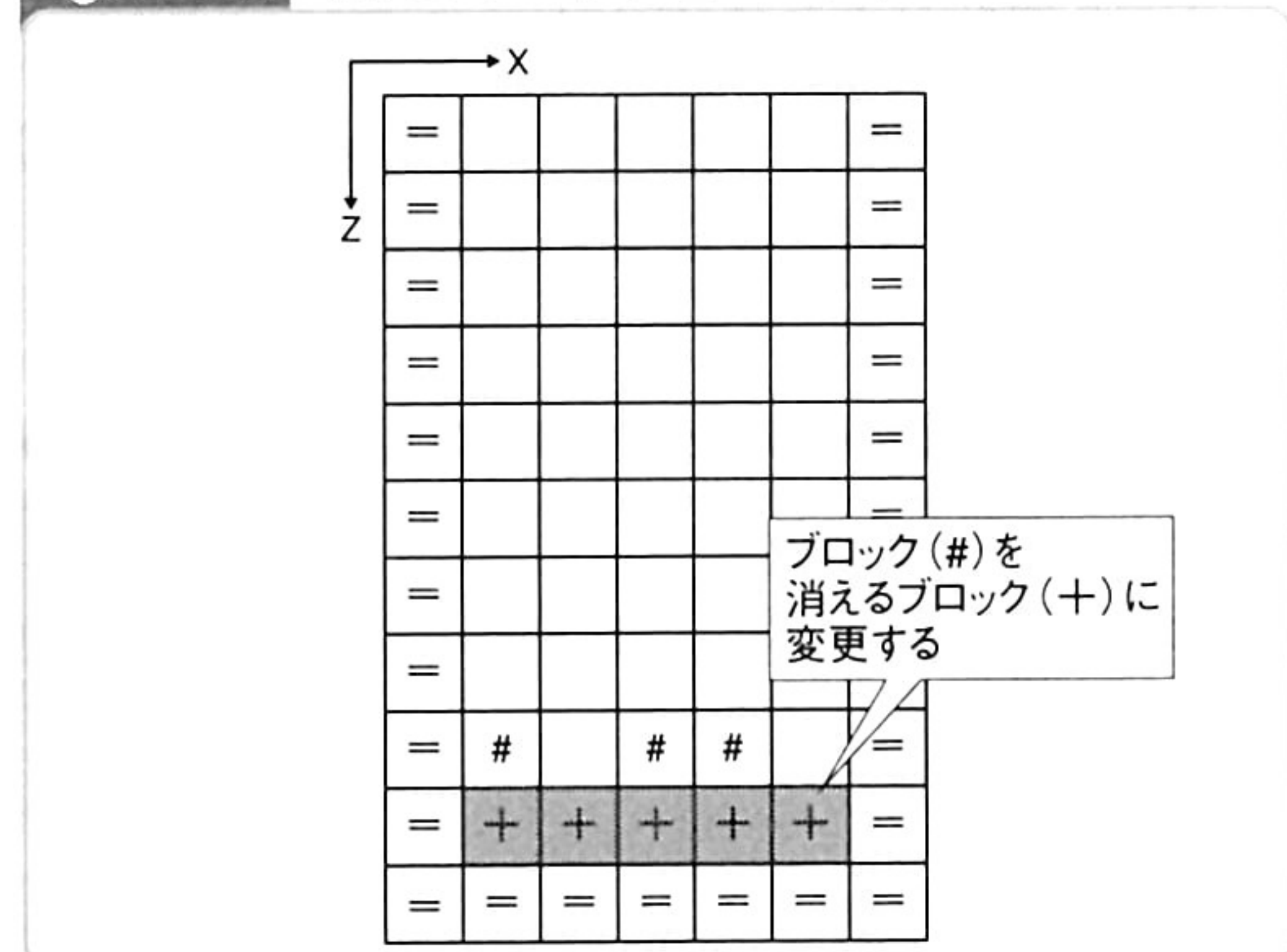
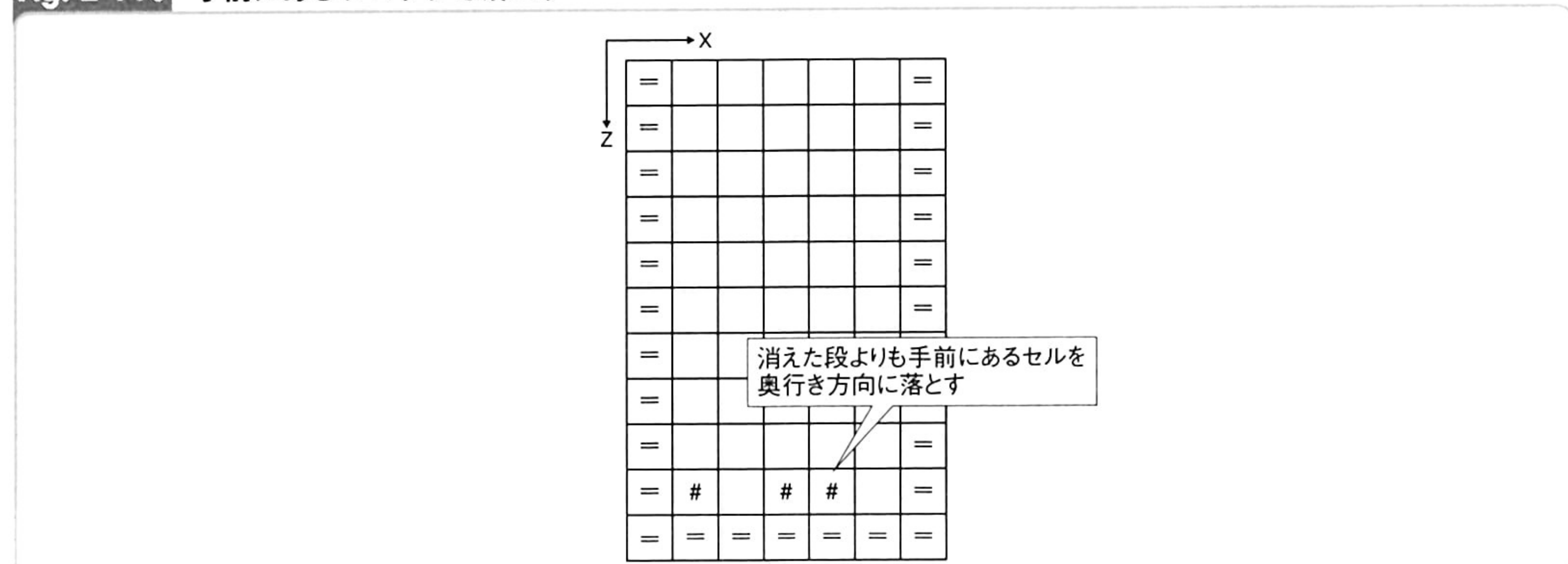



Fig. 2-109 消えるブロックをマークする

ブロック（#）を
消えるブロック（+）に
変更する

Fig. 2-110 手前にあるブロックを落とす

消えた段よりも手前にあるセルを
奥行き方向に落とす

プログラム

List 2-18は3次元のブロックを1段揃えて消すプログラムです。ブロックの移動処理を掲載しました。

ブロックの移動処理は、ブロックが着地したときの処理（着地状態）、1段揃ったブロックを消す処理（消去状態）、そして新しいブロックを出現させる処理（再出現状態）に分かれています。

着地状態では、ステージ全体のセルを調べて、ブロックが隙間なく揃っている段を探します。揃った段が見つかったら、ブロックのセル（#）を、消えるブロックのセル（+）に変更します。そして、タイマーを設定して、消去状態に移行します。揃った段がなければ、再出現状態に移行します。

消去状態では、タイマーを更新します。タイマーが一定値に達したら、消えるブロックの段を探して消し、より手前にあるすべての段を落下させます。そして、再出現状態に移行します。

再出現状態では、新しいブロックをステージ手前に出現させます。あとは「3次元のブロックを落とす」(→p. 123) の入力状態に戻って、同様の処理を繰り返します。

ブロックの描画処理では、ステージのセルを調べて、ブロックと壁を描画します。消えるブロックに関しては、タイマーを使って、だんだん薄くなって消えるように、半透明で描画しています。具体的な処理は、CDroppingBlock3DBlockクラスのDraw関数（付録CD-ROMの「Puzzle￥Stage2.h」）を参照してください。

List 2-18 3次元のブロックを1段揃えて消す（CDroppingBlock3DBlockクラス）

```
// ブロックの移動処理
bool CDroppingBlock3DBlock::Move(const CInputState* is) {

    // ...（中略）...

    // 着地状態
    if (State==2) {

        // 消えるブロックがまったくない場合には、
        // 再出現状態に移行する
        State=4;

        // ブロックが隙間なく揃った段を探す
        for (int z=1; z<DROPPING_BLOCK3D_FIELD_ZSIZE-1; z++) {
            int x, y;
            for (y=1; y<DROPPING_BLOCK3D_FIELD_YSIZE-1; y++) {
                for (x=1; x<DROPPING_BLOCK3D_FIELD_XSIZE-1; x++) {

                    // 空のセルがある場合にはループを抜け出す
                    if (StageCell->Get(x, y, z)==' ') {
```

```
                    y=DROPPING_BLOCK3D_FIELD_YSIZE;
                    break;
                }
            }
        }

        // ブロックが揃った段がある場合の処理
        if (y==DROPPING_BLOCK3D_FIELD_YSIZE-1) {
            for (y=1; y<DROPPING_BLOCK3D_FIELD_YSIZE-1; y++) {
                for (x=1; x<DROPPING_BLOCK3D_FIELD_XSIZE-1; x++) {

                    // 段のブロックを消えるブロック(+)にする
                    StageCell->Set(x, y, z, '+');
                }
            }

            // 消去状態に移行する
            Time=0;
            State=3;
        }
    }
}

// 消去状態
if (State==3) {

    // タイマーの更新
    Time++;

    // タイマーが一定値に達したら、ブロックを消す
    if (Time==20) {

        // 消えるブロック(+)がある段を探す
        for (int i=1; i<DROPPING_BLOCK3D_FIELD_ZSIZE-1; i++) {

            // 消えるブロックが見つかったときの処理
            if (StageCell->Get(1, 1, i)=='+') {

                // 消える段よりも手前にあるすべての段を、
                // 1段ずつ下に落とす
                for (int z=i; z>0; z--) {
                    for (
                        int y=1;
                        y<DROPPING_BLOCK3D_FIELD_YSIZE-1;
                        y++
                    ) {
                        for (
                            int x=1;
                            x<DROPPING_BLOCK3D_FIELD_XSIZE-1;
```

```
                    x++
                ) {
                    StageCell->Set(
                        x, y, z, StageCell->Get(x, y, z-1));
                }
            }
        }
    }
}

            //再出現状態に移行する
            State=4;
        }
    }

    //再出現状態
    if (State==4) {

        //新しいブロックを出現させる
        Init();
    }

    return true;
}
```

まとめ

本章では「落とす」アクションに注目して、いわゆる「落ち物パズルゲーム」のいろいろなパターンを紹介しました。物体を落として積み上げる、というルールは共通していますが、物体を消すための方法や、消した後の効果などは、ゲームによって大きく違います。面白い落ち物パズルゲームを作るために大事なことは、独創的なルールを考え出して、魅力的なゲームバランスを生み出すことです。

というわけで、「落とすゲームは独創性が決め手！」というのが本章のまとめです。

Stage 03

つなぐ

Connect

「つなぐ」というアクションも、パズルゲームに多く採用されています。いろいろな形をした線路やパイプをつなぐゲームが代表例ですが、ものをつないで形を作ったり、線をつないで囲んだりといったゲームもあります。少し変わったものとして、一筆書きの要領でアイテムを回収するゲームや、文字をつないで言葉を作るゲームなども紹介します。

線路をつなぐ

線路をつないでルートを作るアクションです。15パズルの要領で線路をスライドさせて、ルートが途切れないように線路を並べていきます。線路に沿って進んでいるキャラクターが線路から落ちないように、素早くルートを作る必要があります。

ステージには線路が格子状に並んでいます（Fig. 3-1）。そのなかに、1か所だけ線路がない部分があります。これはプレイヤーが自由に動かすことができるカーソルです。

カーソルはレバーで上下左右に動きます（Fig. 3-2）。カーソルを動かすと、線路の配置を変えることができます。カーソルの移動先にある線路が、カーソルの移動元に移動します。この操作を繰り返すと、ステージの配置を最初とはまったく違ったものにすることも可能です。

線路にはいろいろな形があります（Fig. 3-3）。左右に曲がるもの、まっすぐなもの、十字に交差したものなどです。十字に交差した線路は、縦にも横にも通過することができます。

線路をつなぐゲームには『ガッタンゴットン』などがあります。このゲームでは、格子状に配置された線路の上を、機関車が走行しています。カーソルを動かすと、線路をスライドさせて配置を変えることができます。線路を上手に動かしてルートを作り、機関車を乗客のところへ誘導することが、ゲームの目的です。機関車が線路から落ちたり、敵キャラクターにつかまったりするとミスになります。

Fig. 3-1 格子状に並んだ線路

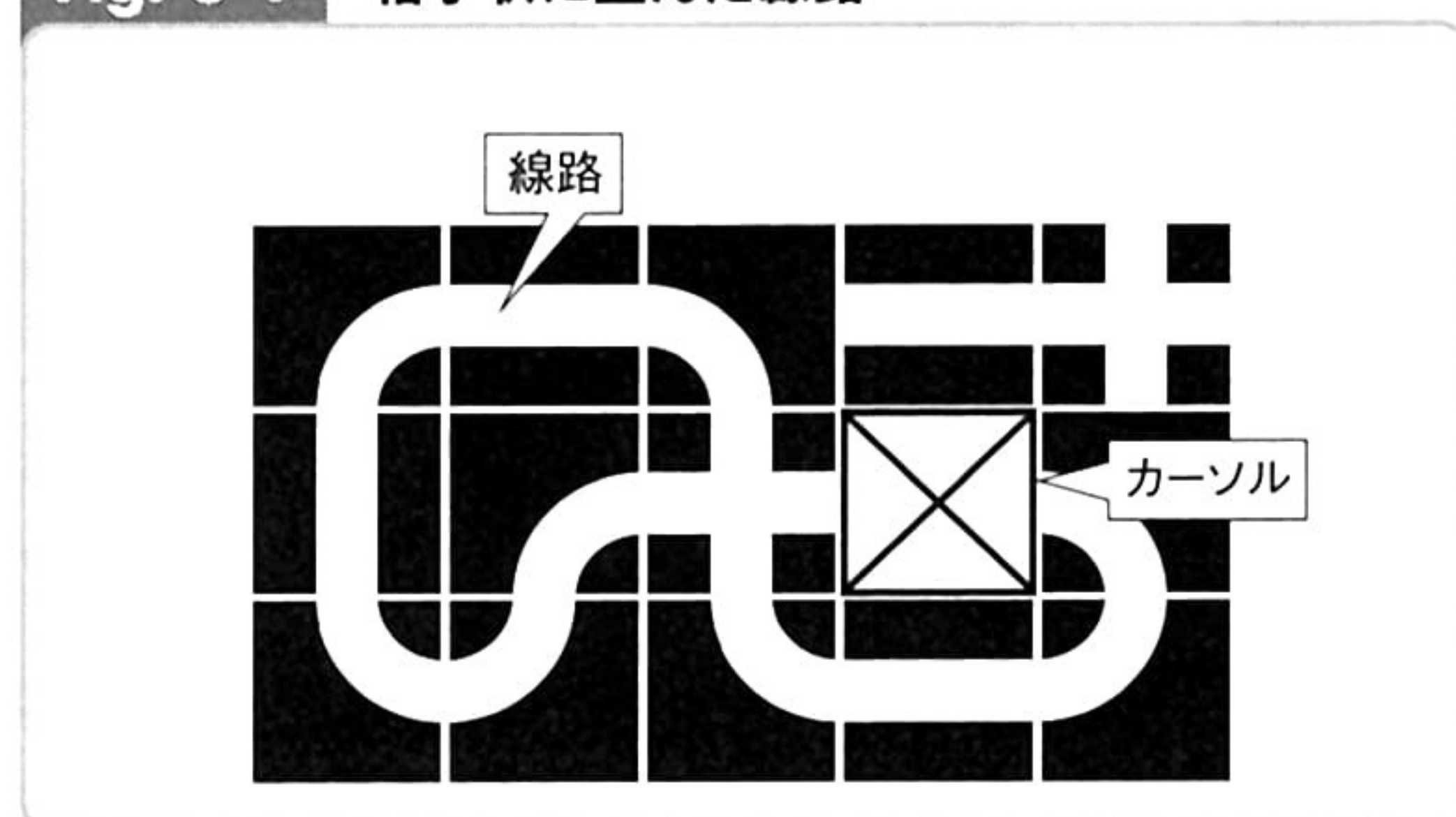



Fig. 3-2 カーソルを動かす

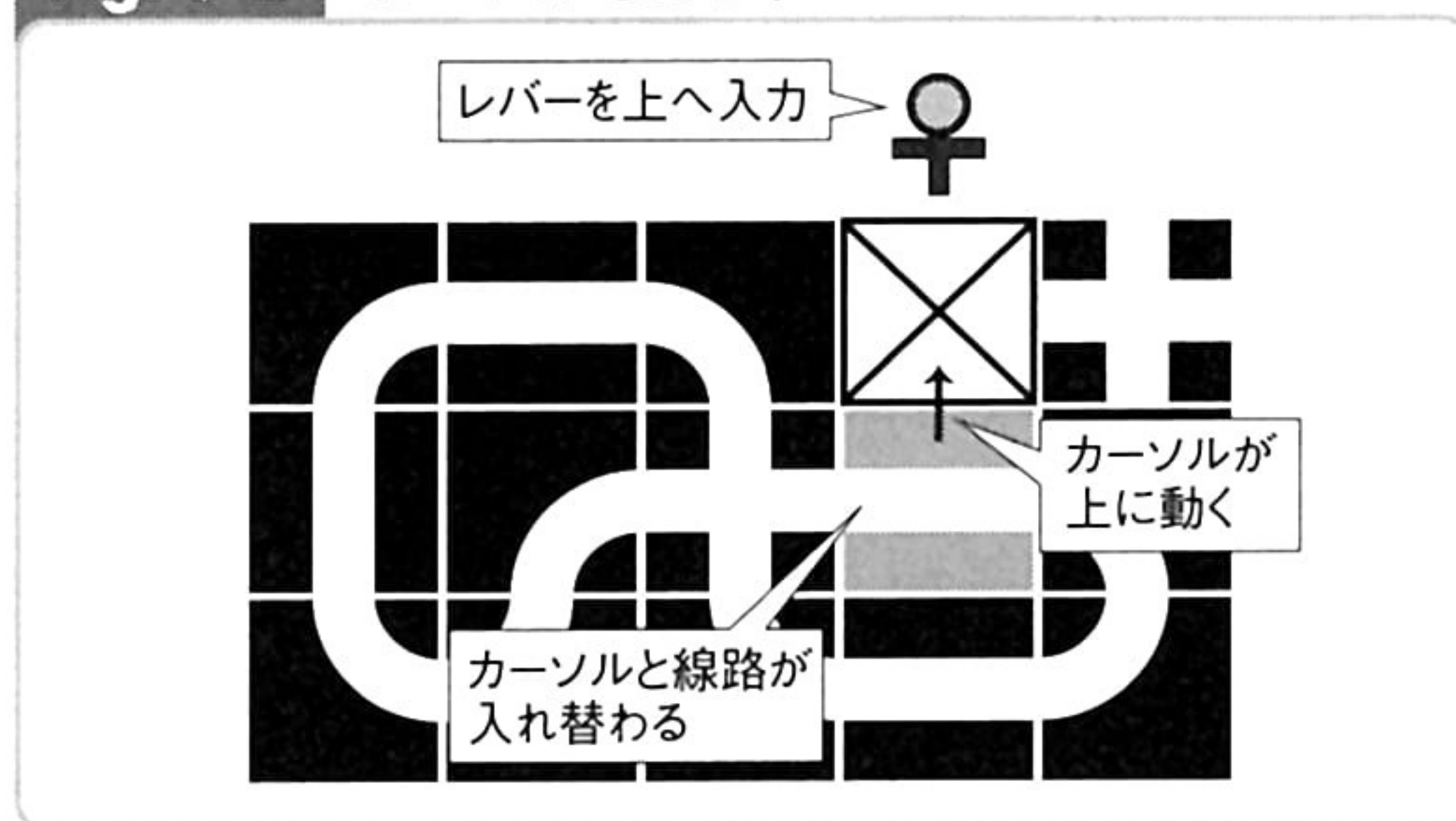
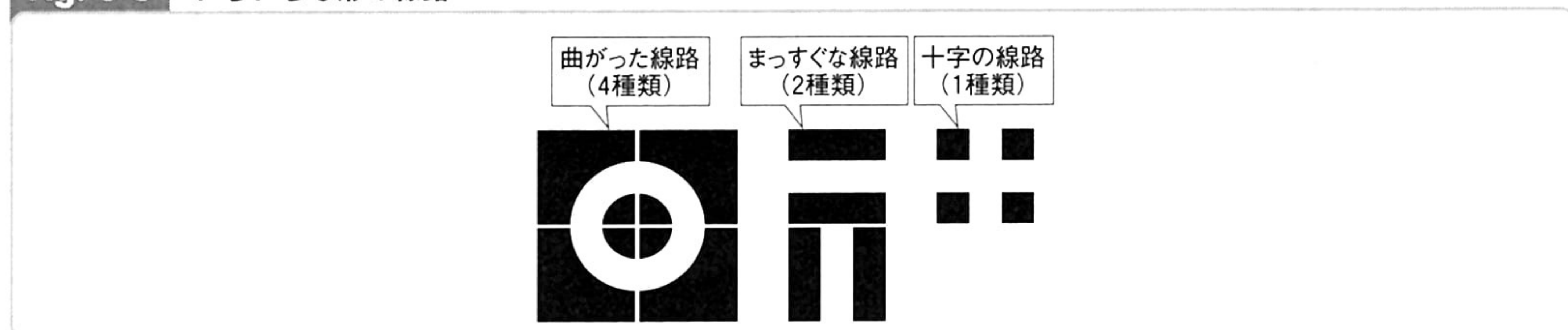



Fig. 3-3 いろいろな形の線路

『キューブリック』も線路をつなぐゲームです。このゲームでは、ステージごとに線路がいろいろな形に配置されており、問題を解いていく感覚でステージをクリアしていきます。

アルゴリズム

線路をつなぐアクションを実現するには、ステージをセルで表現します。セルについては「迷路を歩く」(→p. 10) を参照してください。

まずは線路をセルで表現します (Fig. 3-4)。線路にはいろいろな形があるので、形状ごとに異なる数字を割り振ります。ここでは7種類の形を考えて、それぞれ数字の「1〜7」を割り振ることにしました。カーソルは数字の「0」で表します。

次に、これらの線路を並べてステージを表現します (Fig. 3-5)。ここでは5×3＝15個のセルを使ってステージを表すことにしました。15個のセルのうち、1個のセルはカーソルで、残り14個のセルは線路です。

プレイヤーがレバーを入力したら、カーソルを動かして、線路の配置を変えます。これはカーソルと線路のセルを入れ替えることによって実現します (Fig. 3-6)。例えばレバーを上に入

Fig. 3-4 線路をセルで表現する

Fig. 3-5 ステージをセルで表現する

1	5	2	5	7
6	1	7	0	2
4	3	4	5	3

Fig. 3-6 カーソルの移動

1	5	2	5	7
6	1	7	0	2
4	3	4	5	3

1	5	2	0	7
6	1	7	5	2
4	3	4	5	3

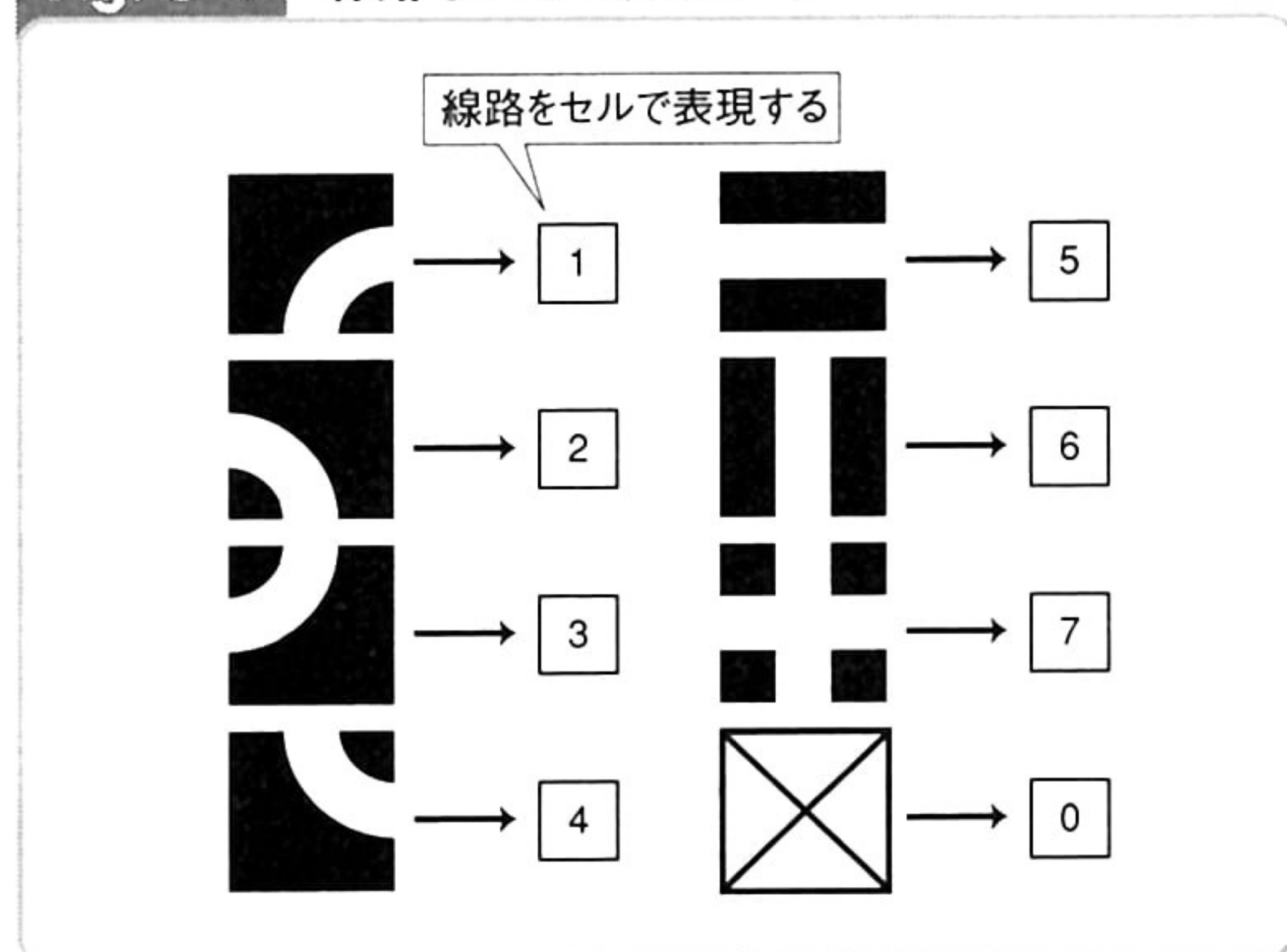
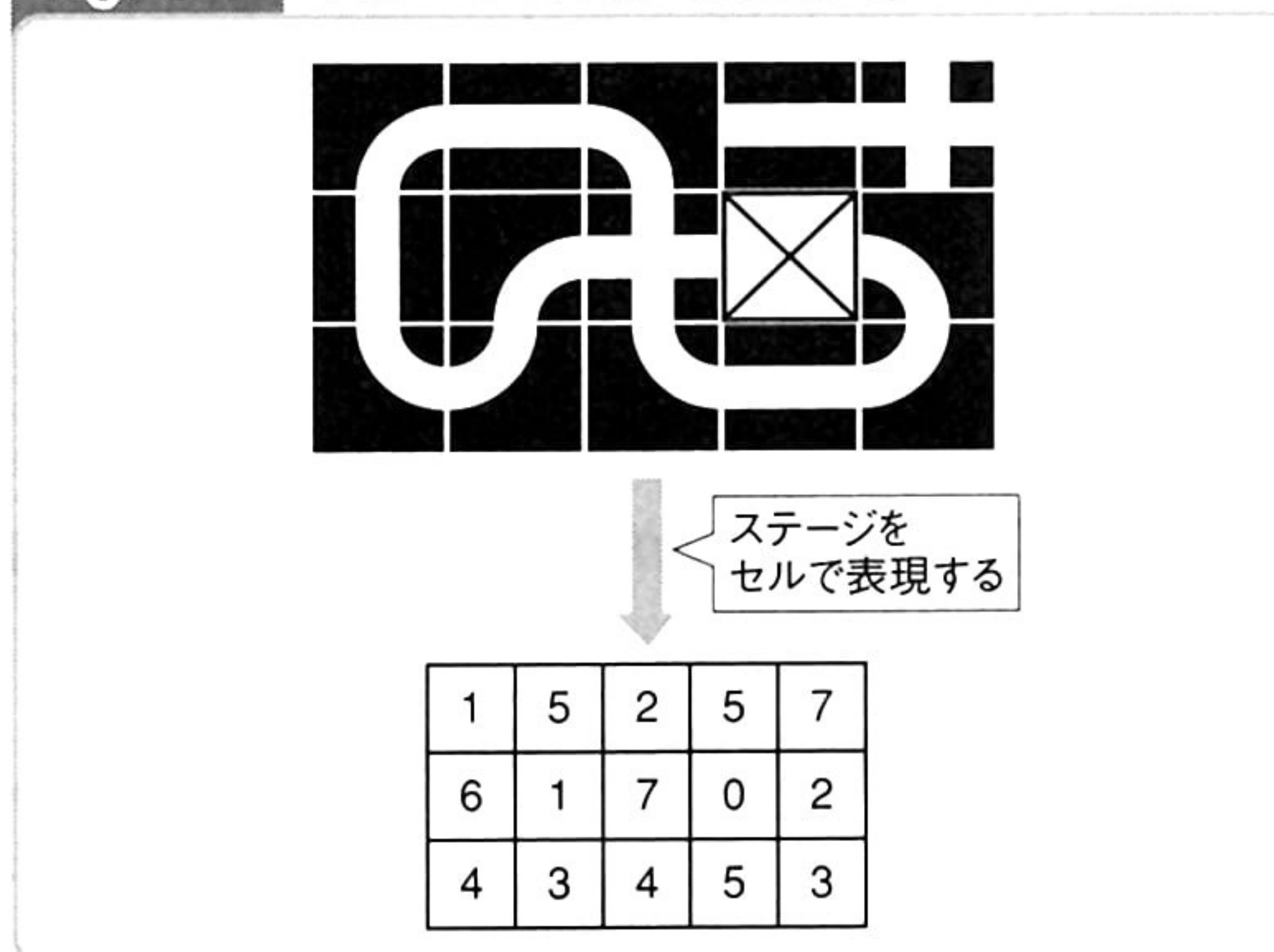
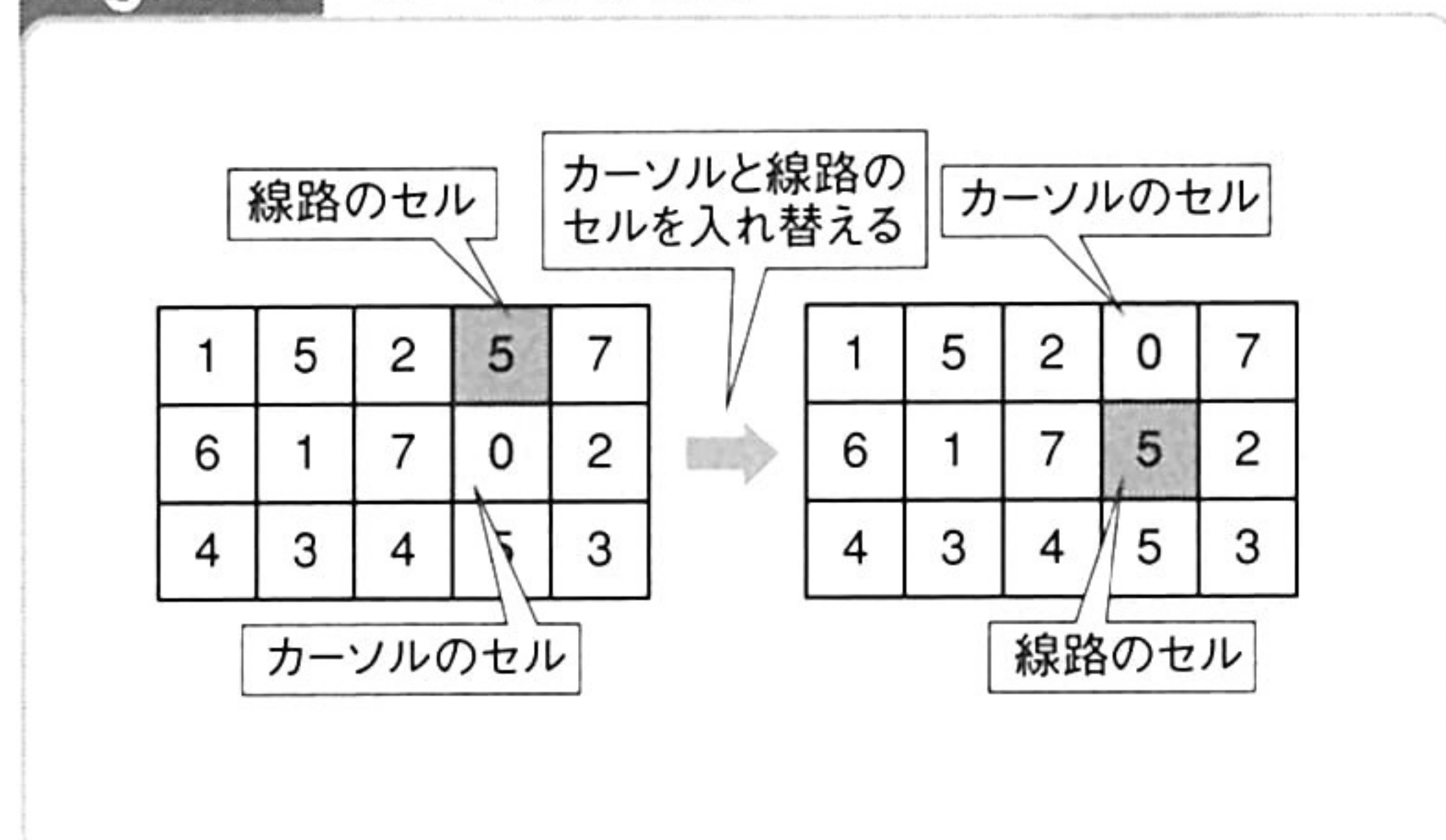
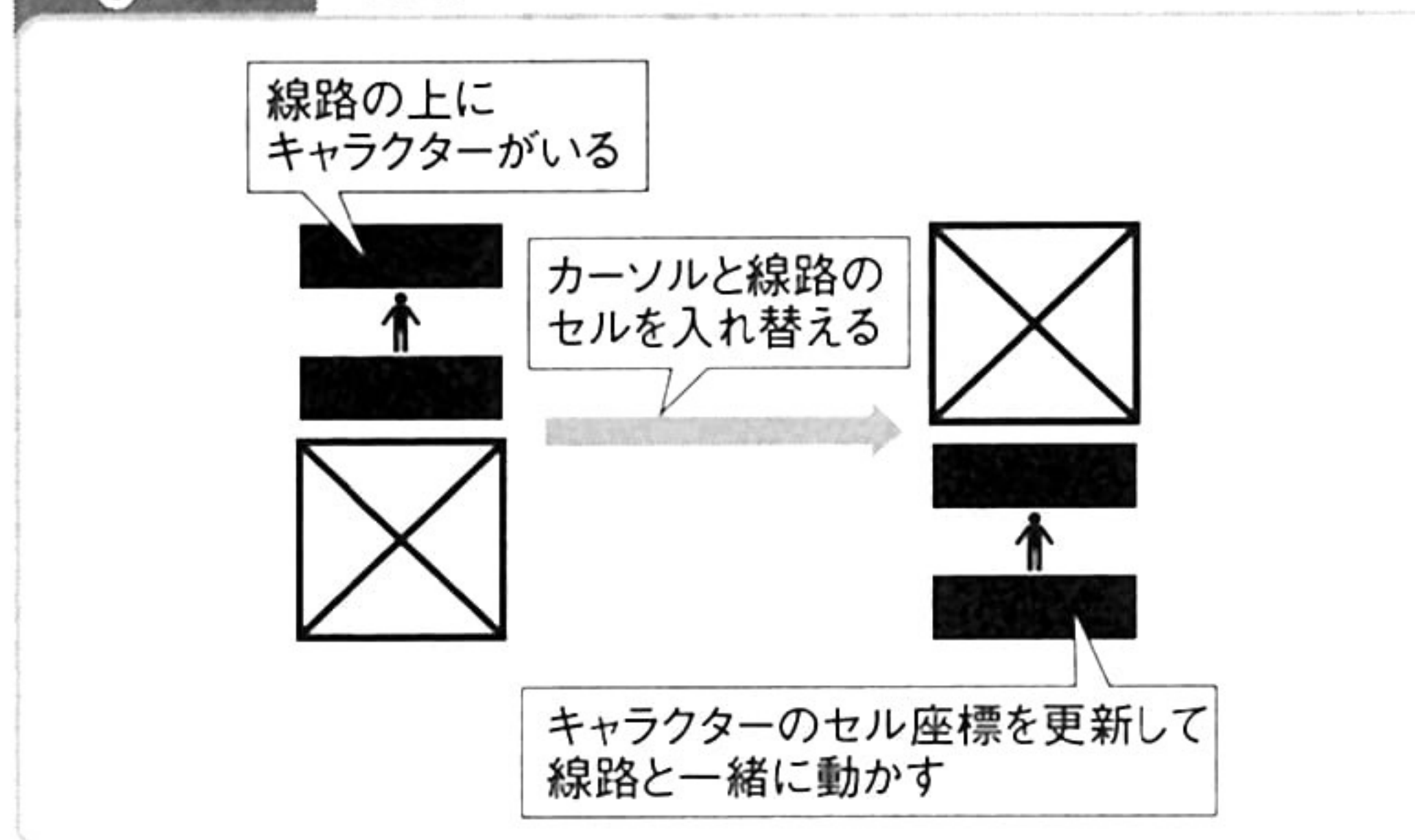



Fig. 3-7 線路の上にキャラクターがいる場合

力したら、カーソルのセルと、カーソルの上にある線路のセルを入れ替えます。

なお、線路の上にキャラクターがいる場合には、キャラクターごと線路を動かします（Fig. 3-7）。カーソルと線路のセルを入れ替えるのと同時に、キャラクターのセル座標（セル単位の座標）を更新します。キャラクターの詳細については「線路に沿って進むキャラクター」（→p. 147）で解説します。

カーソルを動かす

画面上でカーソルを滑らかに動かすためには、タイマーを使って、カーソルの描画座標を少しずつ変化させます（Fig. 3-8）。例えばカーソルが（X, Y）から（X, Y-1）まで移動するときには、

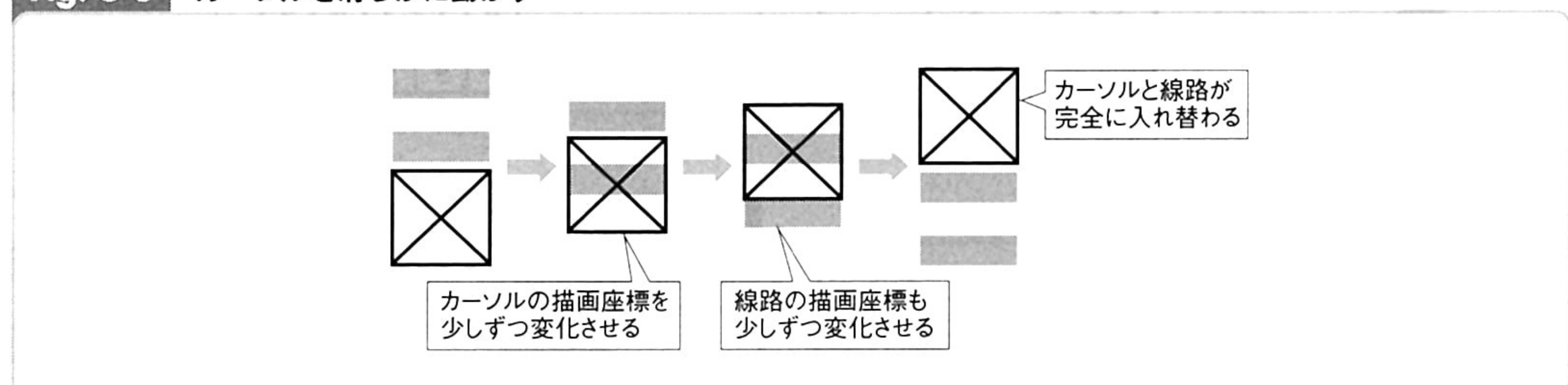
（X, Y-0.1）
（X, Y-0.2）
（X, Y-0.3）
⋮

のように、ある程度の時間をかけて、描画座標を目標値に近づけます。

線路の描画座標についても、カーソルと同様に少しずつ変化させます。移動方向はカーソルとは逆です。例えばカーソルが（X, Y）から（X, Y-1）まで移動するとき、線路の座標は、

（X, Y-0.9）
（X, Y-0.8）
（X, Y-0.7）
⋮

のように変化します。カーソルと線路はすれ違い、一定時間が経過すると位置が入れ替わります。

Fig. 3-8 カーソルを滑らかに動かす

プログラム

List 3-1は線路をつなぐプログラムです。ステージの初期化処理・移動処理・描画処理を掲載しました。

初期化処理（Init関数）は、ステージのセルを初期化して、線路を配置します。線路は数字の「1～7」で表しています。線路のパターンは自由に変更できるので、いろいろと試してみてください。

移動処理（Move関数）では、レバーの入力に応じてカーソルを動かします。ここではカーソルの状態を入力状態と移動状態に分けています。

入力状態では、レバーの入力方向に応じて、カーソルの移動方向を設定します。レバーの入力があったときには、タイマーを設定して、移動状態に移行します。

移動状態では、タイマーを更新します。タイマーが一定値になったら、カーソルと線路のセルを入れ替え、カーソルのセル座標を更新します。線路の上にキャラクターがいる場合には、キャラクターのセル座標も更新します。

描画処理（Draw関数）では、線路とカーソルを描画します。ステージのセルを調べて、線路やカーソルのグラフィックを格子状に並べて表示します。

カーソルが移動状態ならば、タイマーを使って、カーソルの描画座標を少しずつ変化させながら描画します。カーソルと入れ替わる線路については、カーソルとは逆の方向に描画座標を変化させながら表示します。

List 3-1　線路をつなぐ（CConnectedRailStageクラス）

```
// ステージの初期化処理
void CConnectedRailStage::Init() {

    // セルを初期化する
    Cell->Init(
        "152152"
        "457366"
        "157213"
        "453430"
    );

    // カーソルの座標を設定する
    CX=Cell->GetXSize()-1;
    CY=Cell->GetYSize()-1;

    // 入力状態に移行する
    State=0;

    // キャラクターを生成する
    Man=new CConnectedRailMan(Cell, 1, 0, 0);
}
```

```
// ステージの移動処理
bool CConnectedRailStage::Move(const CInputState* is) {

	// 入力状態の処理
	if (State==0) {

		// レバーを左に入力していて、
		// カーソルがステージの左端でなければ、
		// カーソルを左に移動させる
		if (is->Left && CX>0) {

			// 移動方向とタイマーを設定し、
			// 移動状態に移行する
			VX=-1;
			VY=0;
			Time=0;
			State=1;
		}

		// レバーを右に入力していて、
		// カーソルがステージの右端でなければ、
		// カーソルを右に移動させる
		if (is->Right && CX<Cell->GetXSize()-1) {

			// 移動方向とタイマーを設定し、
			// 移動状態に移行する
			VX=1;
			VY=0;
			Time=0;
			State=1;
		}

		// レバーを上に入力していて、
		// カーソルがステージの上端でなければ、
		// カーソルを上に移動させる
		if (is->Up && CY>0) {

			// 移動方向とタイマーを設定し、
			// 移動状態に移行する
			VX=0;
			VY=-1;
			Time=0;
			State=1;
		}

		// レバーを下に入力していて、
		// カーソルがステージの下端でなければ、
		// カーソルを下に移動させる
```

```
		if (is->Down && CY<Cell->GetYSize()-1) {

			// 移動方向とタイマーを設定し、
			// 移動状態に移行する
			VX=0;
			VY=1;
			Time=0;
			State=1;
		}
	}

	// 移動状態の処理
	if (State==1) {

		// タイマーの更新
		Time++;

		// タイマーが一定値に達したら、
		// カーソルと線路の位置を入れ替える
		if (Time==10) {

			// 入れ替える線路の上にキャラクターがいたら、
			// キャラクターも新しい位置にする
			if (Man->CX==CX+VX && Man->CY==CY+VY) {
				Man->CX=CX;
				Man->CY=CY;
			}

			// カーソルと線路のセルを入れ替える
			Cell->Swap(CX, CY, CX+VX, CY+VY);

			// カーソルの座標を更新する
			CX+=VX;
			CY+=VY;

			// 入力状態に移行する
			State=0;
		}
	}

	return true;
}

// ステージの描画処理
void CConnectedRailStage::Draw() {

	// ...（中略）...

	// セルの個数と画面の解像度から、
```

```
	// 描画サイズを決める
	int xs=Cell->GetXSize(), ys=Cell->GetYSize();
	float
		sw=Game->GetGraphics()->GetWidth()/xs,
		sh=Game->GetGraphics()->GetHeight()/(ys+1);

	// ステージのすべてのセルについて処理する
	for (int y=0; y<ys; y++) {
		for (int x=0; x<xs; x++) {

			// セルの種類を取得する
			char c=Cell->Get(x, y)-'0';

			// 描画座標の計算
			float
				dx=x, dy=y+1,
				f=Time*0.1f, vx=VX*f, vy=VY*f;

			// 移動状態ならば、
			// カーソルを滑らかに動かすために、
			// 描画座標を調整する
			if (State==1 && x==CX && y==CY) {
				dx+=vx;
				dy+=vy;
			} else

			// 移動状態ならば、
			// 線路を滑らかに動かすために、
			// 描画座標を調整する
			if (State==1 && x==CX+VX && y==CY+VY) {
				dx-=vx;
				dy-=vy;
			}

			// セル座標を画面上の座標に変換する
			dx*=sw;
			dy*=sh;

			// ...（中略）...

			// 線路またはカーソルのグラフィックを描画する
			Game->Texture[TEX_RAIL0+c]->Draw(
				dx, dy, sw, sh,
				0, 0, 1, 1, COL_BLACK);
		}
	}

	// ...（中略）...
}
```

SAMPLE

「CONNECTED RAIL」は「線路をつなぐ」「線路に沿って進むキャラクター」「キャラクターの進路を予測して表示する」のサンプルです。

レバーの上下左右（カーソルキーの上下左右）でカーソルが移動します。カーソルを動かすと、移動先の線路とカーソルの位置を入れ替えることができます。線路の上にキャラクターがいる場合には、キャラクターごと動かすことができます。

キャラクターは線路に沿って動いています。キャラクターが線路から落ちないように、上手に線路を配置して、ルートを作ってください。このサンプルでは、キャラクターが線路から落ちてもミスにはならず、キャラクターはその場で待機します。再びルートがつながると、キャラクターが動き出します。

灰色で表示されているのは、キャラクターの進路予測です。線路の配置が変わるたびに、新しい予測を表示します。

CONNECTED RAIL → p. 387

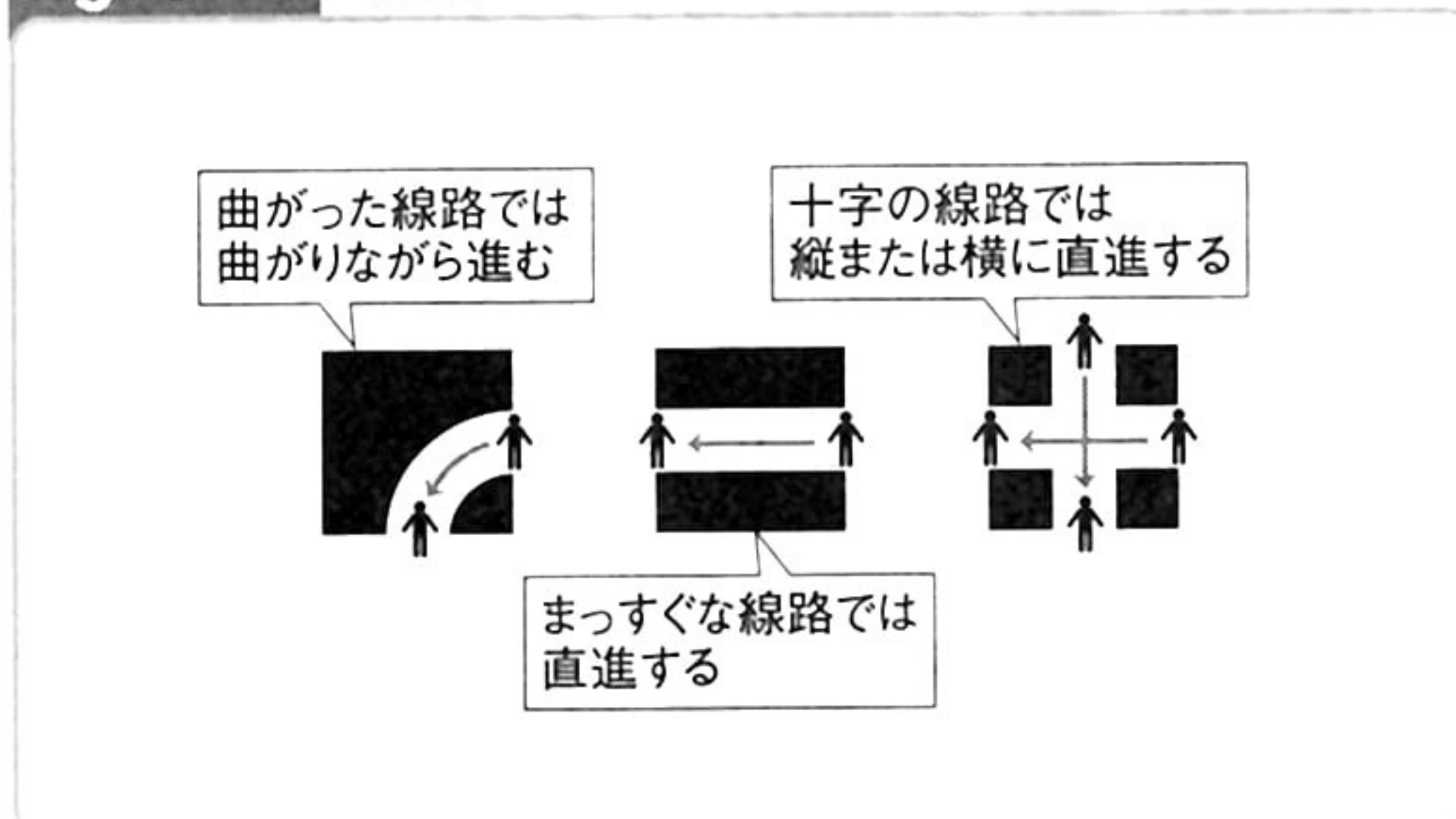
線路に沿って進むキャラクター

ステージに配置された線路の上を進むキャラクターです。プレイヤーが操作をしなくても、キャラクターは自動的に線路に沿って進んでいきます。

線路がつながっているかぎり、キャラクターは線路に沿って進みます（Fig. 3-9）。線路が途切れていると、ゲームによってはキャラクターが線路から落ちてミスとなります。本書のサンプルでは、線路が途切れたところでキャラクターが待機し、線路がつながると再び動き出すことにしています。

キャラクターの動きは、線路の形状に応じて変わります（Fig. 3-10）。曲がった線路の場合には、線路に沿ってキャラクターが曲がります。まっすぐな線路の場合には、キャラクターは直進します。十字の線路の場合には、キャラクターは縦または横に直進します。

Fig. 3-9 線路に沿って進むキャラクター

Fig. 3-10 線路の形状に応じたキャラクターの動き

アルゴリズム

キャラクターを線路に沿って進ませるには、線路を通過したときの、キャラクターの移動方向の変化に注目します。例えば、曲がった線路について考えてみましょう（Fig. 3-11）。キャラクターの移動方向は「0～3」の数値で表します。0が右、1が下、2が左、3が上です。各方向に進むキャラクターが線路に入ってきた場合と、線路から出るときに移動方向がどう変わるのかを考えます。

キャラクターの移動方向が0、つまり右に進んでいるときに線路に入ったとします。線路を通過すると、キャラクターの移動方向は下、つまり1になります。同様に、上（3）に進んで線路に入ったときには、移動方向は左（2）に変わります。

キャラクターが下（1）や左（2）に進んでいるときには、線路に入ることができません。この場合は移動方向を「4（移動不可）」で表すことにしましょう。

以上のように考えると、キャラクターが線路に入るときの移動方向（0～3）に対して、線路から出るときの移動方向を、

{1, 4, 4, 2}

のようなテーブルで表すことができます。例えば、入るときの移動方向が右（0）のときには、0番目の要素（最初の要素）を参照すれば、出るときの移動方向が下（1）だということがわかります。

次に、まっすぐな線路について考えましょう（Fig. 3-12）。線路に入るときの移動方向が右（0）のときには、出るときの移動方向も右（0）です。入るときの移動方向が左（2）の場合も同様に、出るときの移動方向が左（2）になります。

入るときの移動方向が下（1）または上（3）のときには、線路に入ることができないので、出るときの移動方向は移動不可（4）で表します。以上により、まっすぐな線路に関するテーブルは、

{0, 4, 2, 4}

Fig. 3-11 曲がった線路とキャラクターの移動方向

キャラクターの移動方向を0から3で表す
右(0)に入ったキャラクターは下(1)に出る
{1, 4, 4, 2}
移動方向のテーブルを作る
上(3)に入ったキャラクターは左(2)に出る

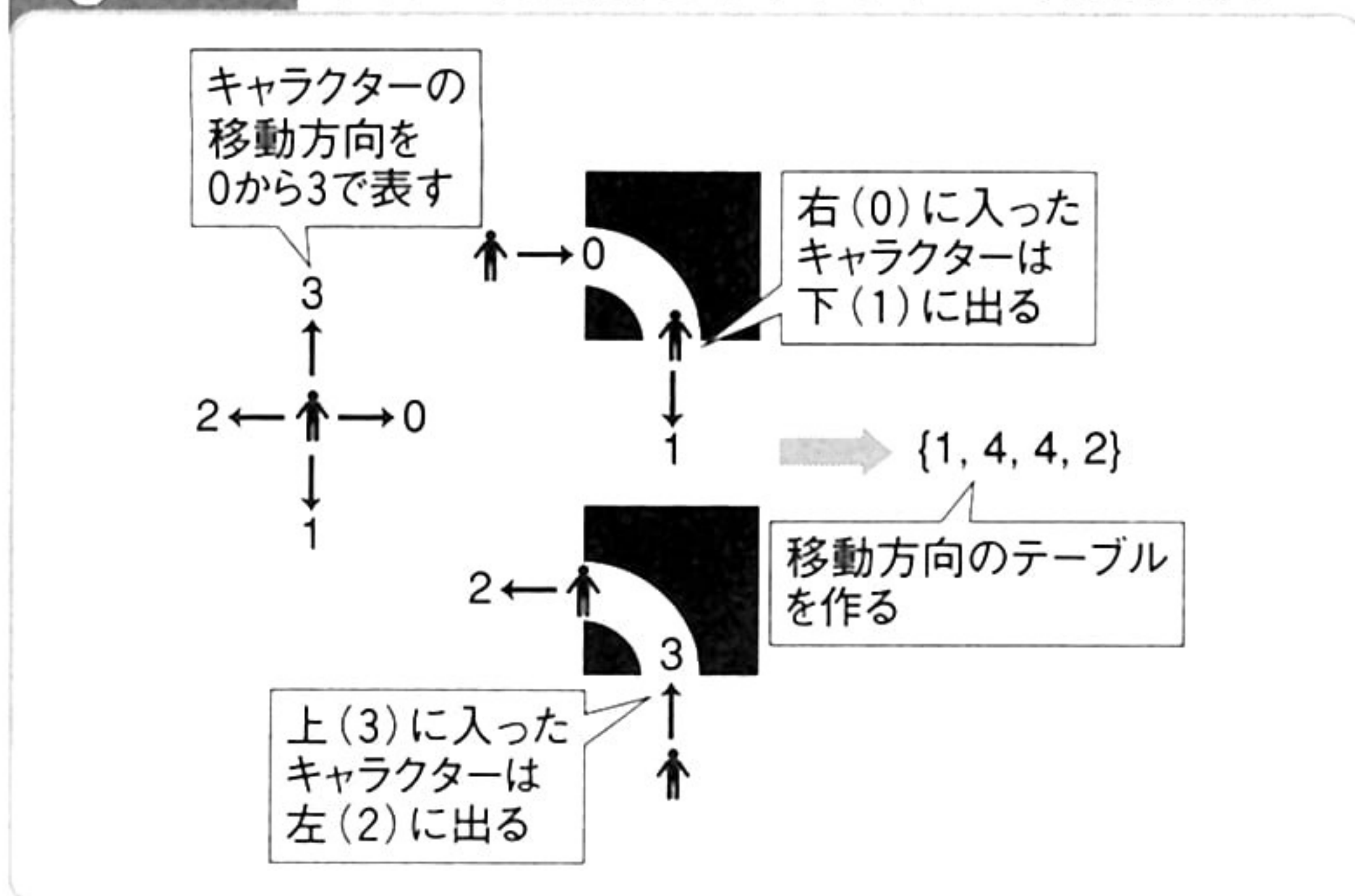



Fig. 3-12 まっすぐな線路とキャラクターの移動方向

右(0)に入ったキャラクターは右(0)に出る
移動方向のテーブルを作る
{0, 4, 2, 4}
左(2)に入ったキャラクターは左(2)に出る

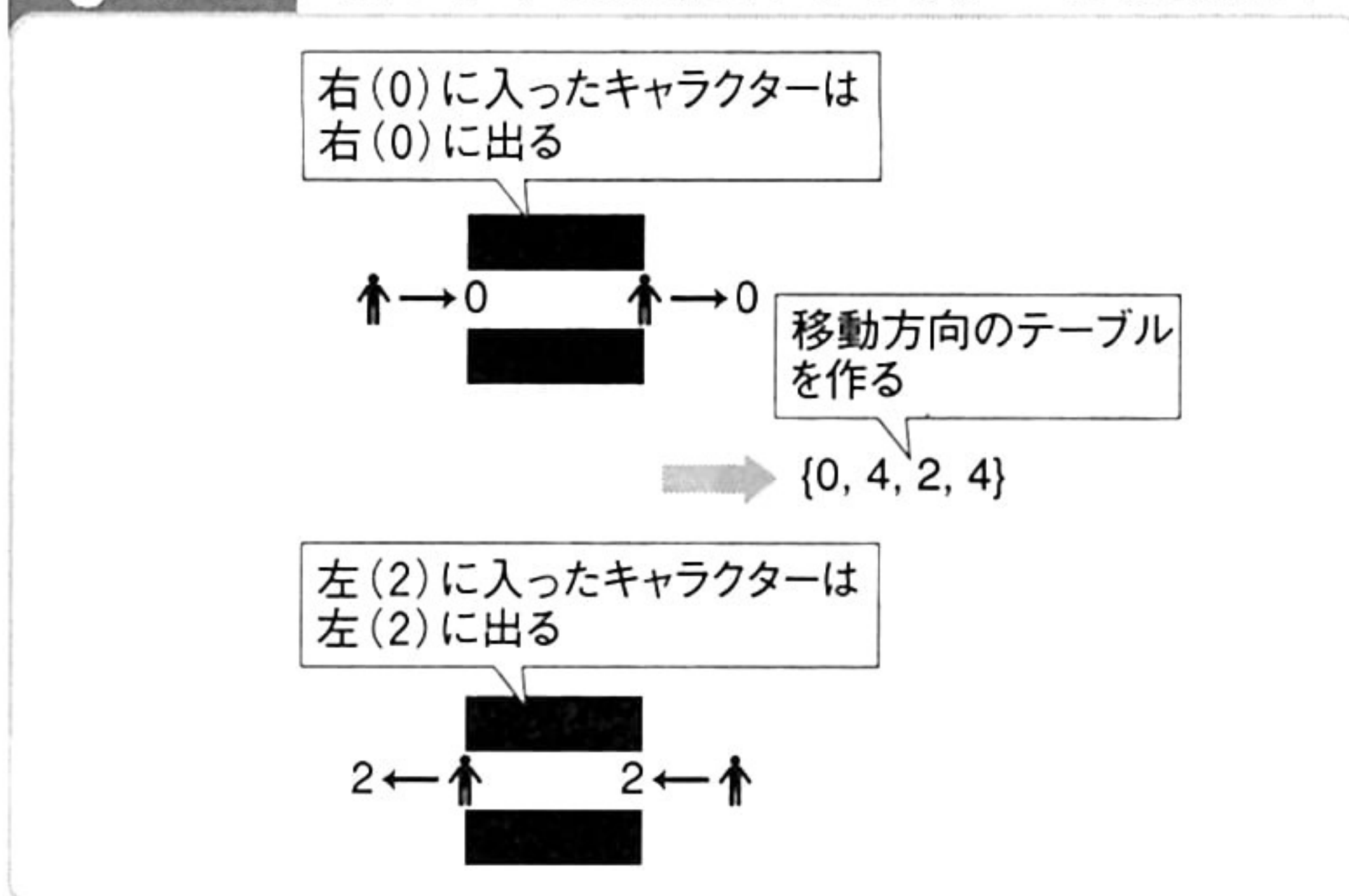



Fig. 3-13 十字の線路とキャラクターの移動方向

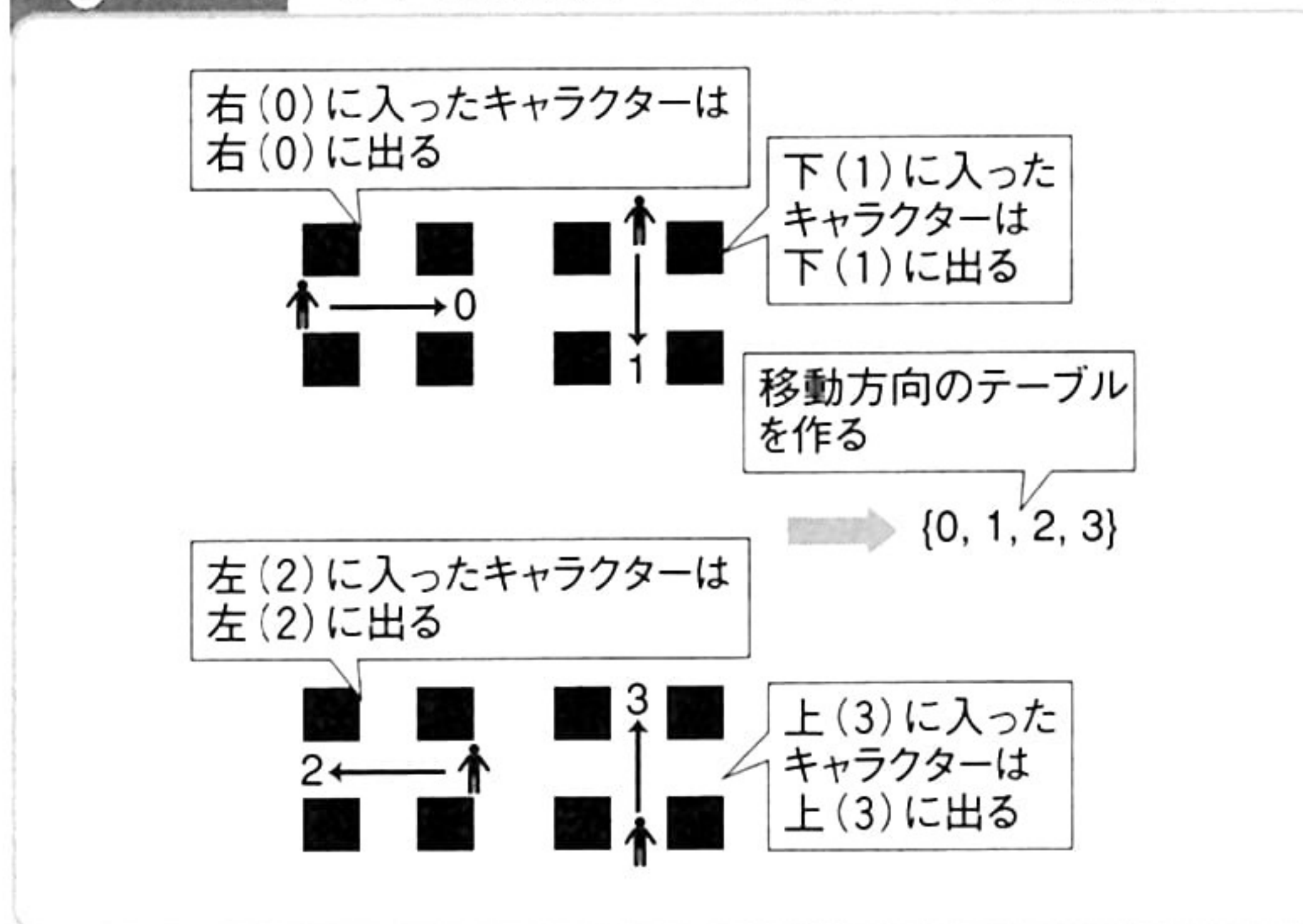



Fig. 3-14 それぞれの線路に関する移動方向のテーブル

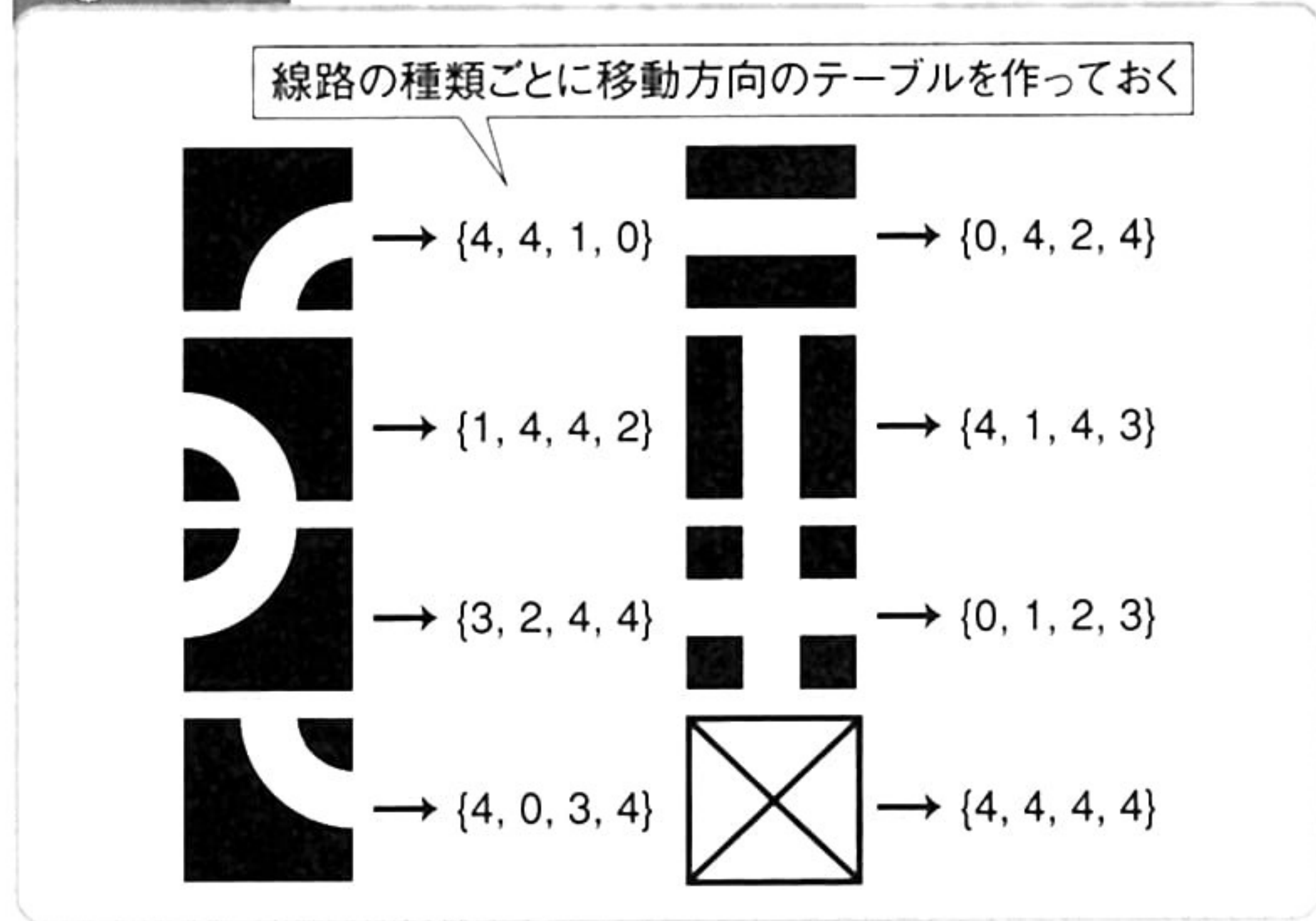



Fig. 3-15 キャラクターが曲がった線路を通る

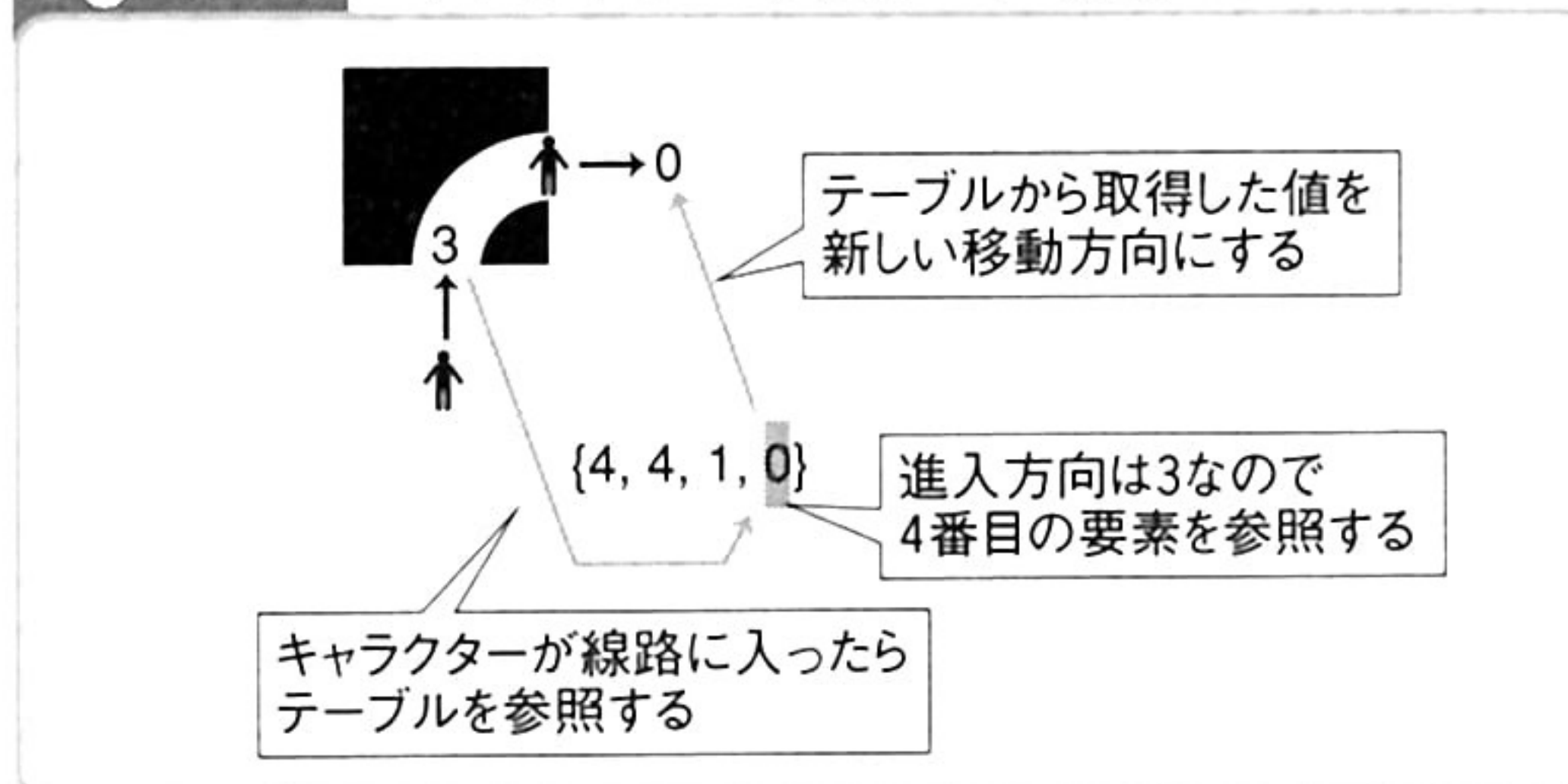
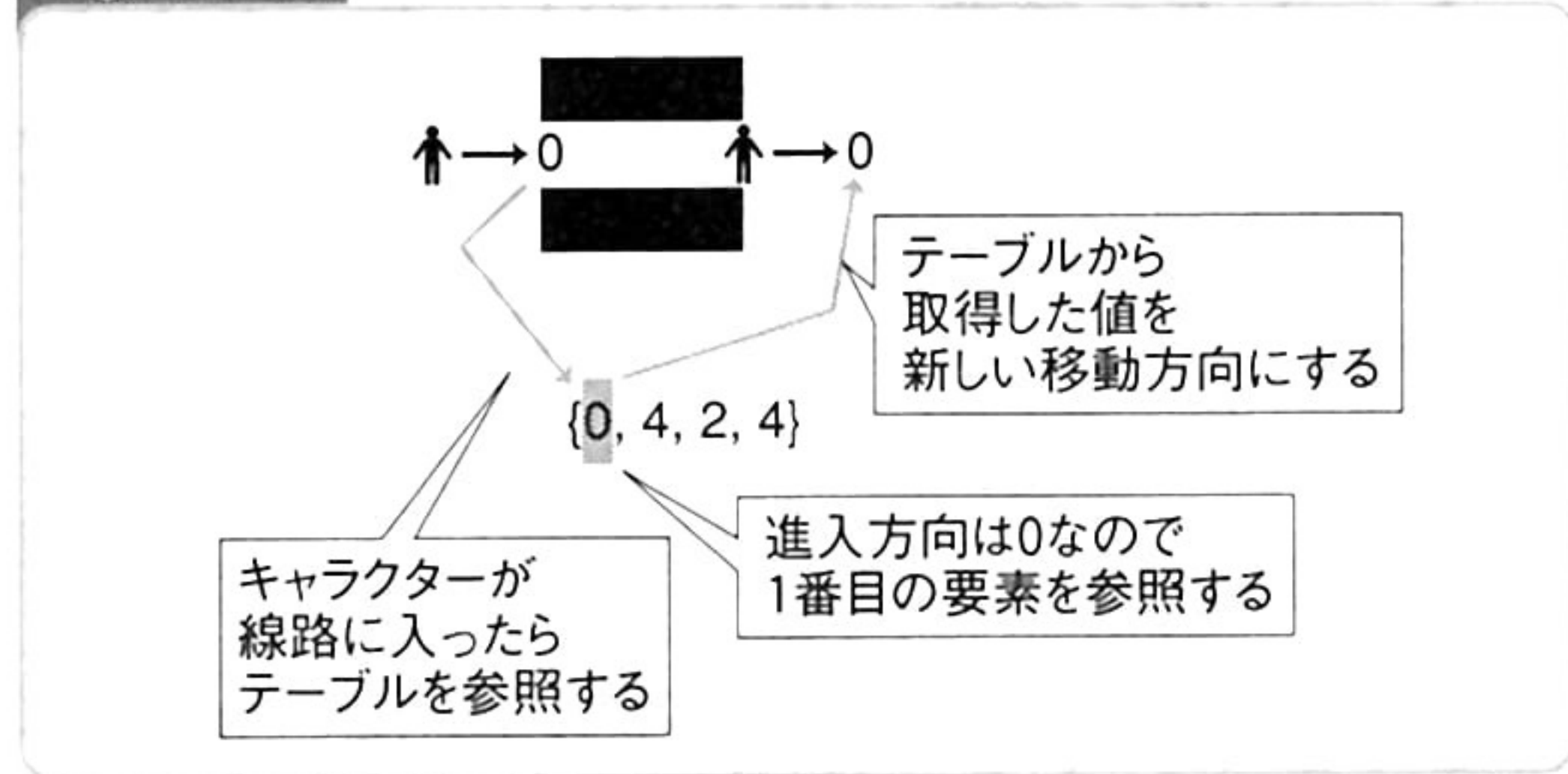



Fig. 3-16 キャラクターがまっすぐな線路を通る

になります。

最後に、十字の線路について考えます（Fig. 3-13）。これはまっすぐな線路を縦横に組み合わせたような性質の線路です。いずれの方向から入ったときにも、その方向のままで出てきます。例えば入るときの移動方向が左（0）の場合、出るときの移動方向も左（0）です。したがって移動方向のテーブルは、

{0, 1, 2, 3}

となります。

線路は全部で7種類あります。曲がった線路が4種類、まっすぐな線路が2種類、十字の線路が1種類です。それぞれの線路について、移動方向のテーブルを作っておきます（Fig. 3-14）。

この移動方向のテーブルを使えば、キャラクターを線路に沿って動かすことができます。キャラクターが線路に入るたびに、テーブルを参照して、キャラクターの移動方向を変化させます。キャラクターが線路に入れないときには、その場で待機させます。

例えば、キャラクターが曲がった線路に入ったら、テーブルにしたがって移動方向を変化させます（Fig. 3-15）。まっすぐな線路を通る場合にも、テーブルを参照して、キャラクターを直進させます（Fig. 3-16）。

滑らかに線路の上を動かす

移動についてもう1つ考える必要があるのは、キャラクターを線路の上で滑らかに動かす方法です。それにはタイマーを使って、キャラクターの描画座標を少しずつ変化させます。

曲がった線路の場合には、セルの隅を中心として、線路に沿って回転するようにキャラクターを動かします（Fig. 3-17）。タイマーをTime、回転の中心を（rx, ry）、開始角度をrad、単位時間あたりの回転角度をvradとすると、キャラクターの描画座標（X, Y）は、

X=rx+cos（rad+vrad×Time）
Y=ry+sin（rad+vrad×Time）

となります。回転の中心座標・開始角度・回転方向については、線路の形状ごとに異なるので、テーブルを作っておくとよいでしょう。なお、角度はラジアンで表します。

まっすぐな線路の場合には、キャラクターをX方向またはY方向に少しずつ直進させます（Fig. 3-18）。タイマーをTime、移動の開始位置を（x, y）、移動速度を（vx, vy）とすると、キャラクターの描画座標（X, Y）は、

X=x+vx×Time
Y=y+vy×Time

となります。十字の線路については、まっすぐな線路と同様です。

Fig. 3-17 曲がった線路でキャラクターを滑らかに動かす

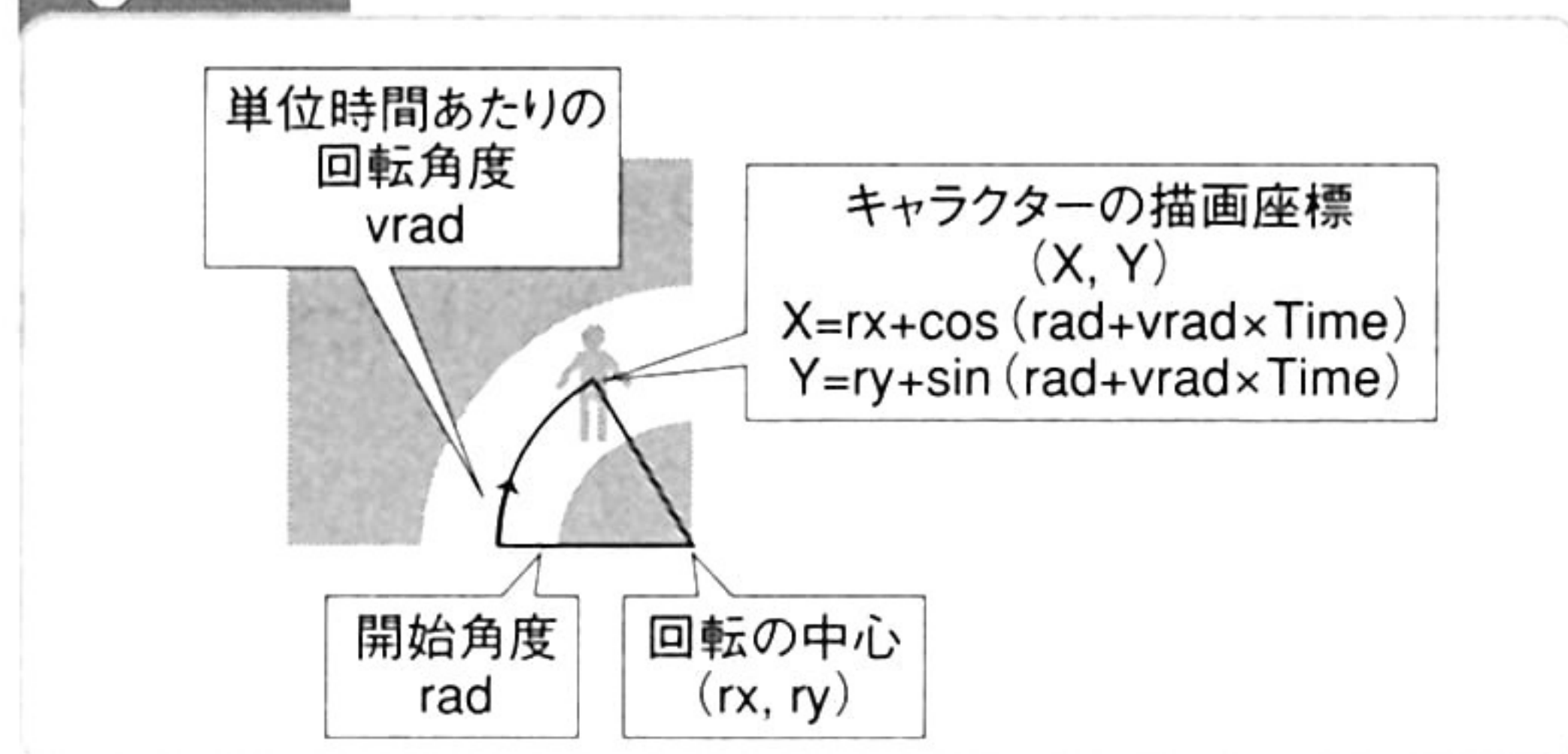
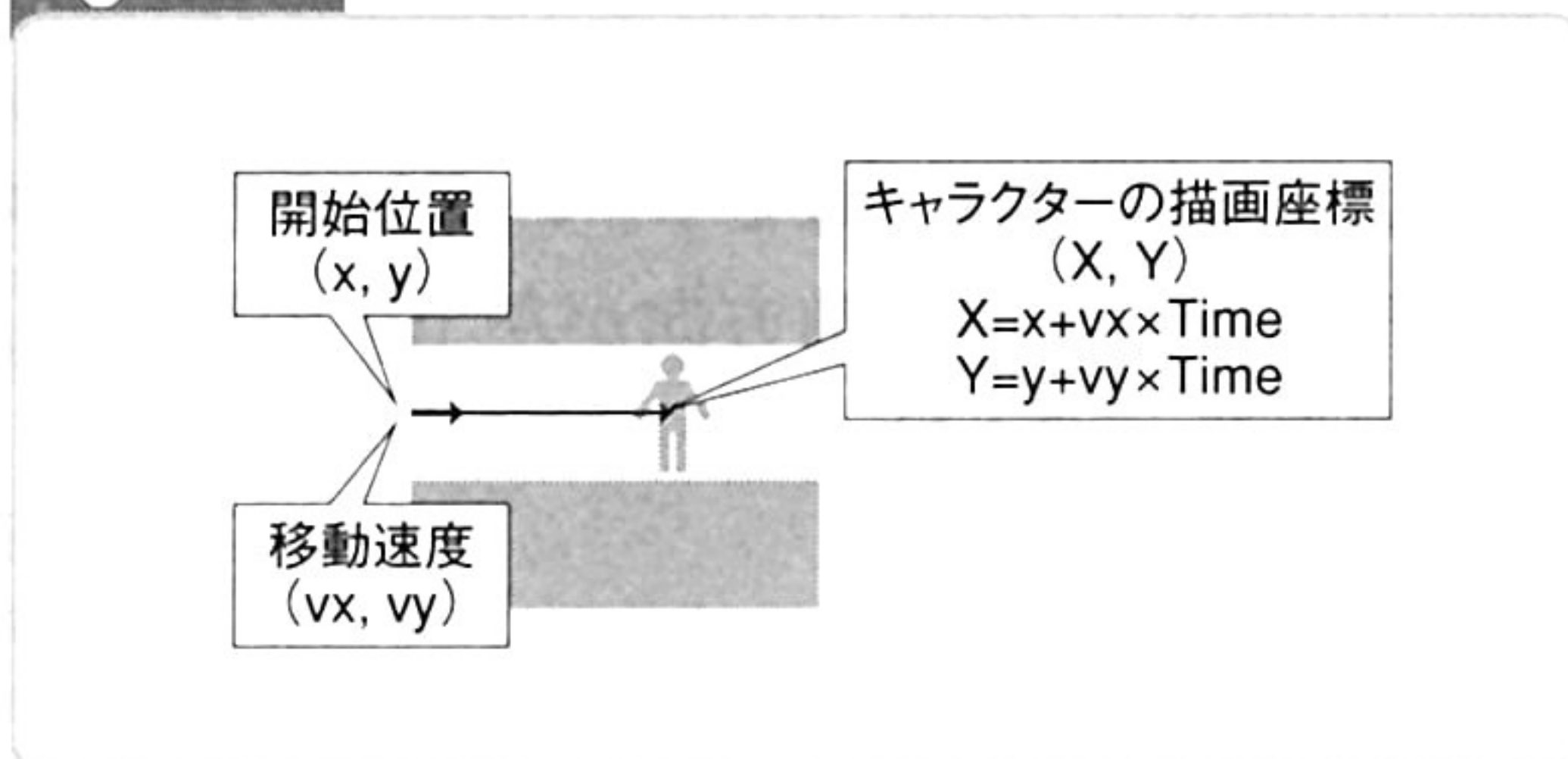



Fig. 3-18 まっすぐな線路でキャラクターを滑らかに動かす

プログラム

List 3-2は線路に沿って進むキャラクターのプログラムです。移動に用いる各種のテーブルと、キャラクターの移動処理、そしてステージの描画処理を掲載しました。キャラクターの描画はステージの描画処理で行います。

キャラクターの移動処理（Move関数）では、キャラクターを滑らかに動かすために、タイマーを更新します。一定時間が経過したら、移動方向のテーブルを使って、キャラクターの移動方向を変化させます。新しい移動方向は、これから入る線路の種類と、線路に入る方向によっ

て決まります。線路に進入できないときには、その場で待機します。

次に、タイマーを使って、キャラクターを画面上で滑らかに動かすための処理を行います。曲がった線路の場合には、線路に沿って回転するように、キャラクターの描画座標を更新します。まっすぐな線路の場合には、線路に沿って直進するように、描画座標を更新します。

ステージの描画処理（Draw関数）では、線路やカーソルと合わせて、キャラクターの描画も行います。キャラクターの描画はキャラクターの描画処理で行う方法もあるのですが、座標計算などに共通部分が多いため、このサンプルではステージの描画処理で行いました。

キャラクターを描画するには、移動処理で計算した描画座標を使ってキャラクターのグラフィックを表示します。動いている線路の上にキャラクターがいるときには、キャラクターが線路と一緒に動くように、描画座標を調整します。

List 3-2　線路に沿って進むキャラクター（CConnectedRailManクラス、CConnectedRailStageクラス）

```
// キャラクターを進行させるための定数
static const int

    // キャラクターが1個のセルを通過する時間
    CONNECTED_RAIL_TIME=60,

    // 移動方向（0〜3）を、
    // X方向とY方向の座標に変換するためのテーブル
    CONNECTED_RAIL_VX[]={1, 0, -1, 0},
    CONNECTED_RAIL_VY[]={0, 1, 0, -1},

    // 線路に入ったときの移動方向（0〜3）を、
    // 線路から出るときの移動方向に変換するためのテーブル
    // （4はその移動方向から進入できないことを示す）
    CONNECTED_RAIL_DIR[][4]={
        {4, 4, 4, 4},
        {4, 4, 1, 0}, {1, 4, 4, 2},
        {3, 2, 4, 4}, {4, 0, 3, 4},
        {0, 4, 2, 4}, {4, 1, 4, 3},
        {0, 1, 2, 3}
    };

// キャラクターの移動処理
bool CConnectedRailMan::Move(const CInputState* is) {

    // 一定時間が経過するまでは、タイマーを更新する
    if (Time<CONNECTED_RAIL_TIME) {
        Time++;
    } else

    // 一定時間が経過したら、移動方向を更新する
    {
        // 移動先のセル座標を計算する
```

```
		int
			cx=CX+CONNECTED_RAIL_VX[Dir],
			cy=CY+CONNECTED_RAIL_VY[Dir];

		// 移動先がステージ内かどうかを確認する
		if (
			cx>=0 && cx<Cell->GetXSize() &&
			cy>=0 && cy<Cell->GetYSize()
		) {
			// 移動先の線路へ現在の移動方向で進入したときの、
			// 新しい移動方向を取得する
			int dir=CONNECTED_RAIL_DIR[Cell->Get(cx, cy)-'0'][Dir];

			// 線路に進入できるかどうかを調べる
			if (dir!=4) {

				// 進入できるならば、
				// セル座標と移動方向を更新し、
				// タイマーを設定する
				CX=cx;
				CY=cy;
				Dir=dir;
				Time=0;
			}
		}
	}

	// 現在キャラクターがいるセルを取得する
	char c=Cell->Get(CX, CY)-'0'-1;

	// 曲がった線路の場合
	if (0<=c && c<=3) {

		// キャラクターを曲がらせるための定数
		static const int

			// 回転中心の相対座標
			rx[]={1, 0, 0, 1},
			ry[]={1, 1, 0, 0},

			// 回転を開始する角度
			rad[][4]={
				{2, 3, 4, 4}, {4, 3, 0, 4},
				{4, 4, 0, 1}, {2, 4, 4, 1}
			},

			// 回転方向
			vrad[][4]={
				{1, -1, 0, 0}, {0, 1, -1, 0},
```

```
                {0, 0, 1, -1}, {-1, 0, 0, 1}
            };

        // タイマーの値に応じて、
        // 線路に沿って滑らかに曲がるように、
        // キャラクターの描画座標を設定する
        float f=
            (rad[c][Dir]+vrad[c][Dir]*(float)Time/
            CONNECTED_RAIL_TIME)*D3DX_PI/2;
        X=CX+rx[c]-1.0f/6+cosf(f)/2;
        Y=CY+ry[c]-1.0f/6+sinf(f)/2;
    } else

    // まっすぐな線路、または十字の線路の場合
    {
        // タイマーの値に応じて、
        // 線路に沿って滑らかに直進するように、
        // キャラクターの描画座標を設定する
        float f=(float)Time/CONNECTED_RAIL_TIME-0.5f;
        X=CX+1.0f/3+CONNECTED_RAIL_VX[Dir]*f;
        Y=CY+1.0f/3+CONNECTED_RAIL_VY[Dir]*f;
    }

    return true;
}

// ステージの描画処理
void CConnectedRailStage::Draw() {

    // ...（中略）...

    // キャラクターの描画座標
    float dx=Man->X, dy=Man->Y+1;

    // カーソルが移動状態で、
    // キャラクターが動いている線路の上にいる場合には、
    // 線路とともに動くように描画座標を設定する
    if (State==1 && Man->CX==CX+VX && Man->CY==CY+VY) {
        dx-=VX*Time*0.1f;
        dy-=VY*Time*0.1f;
    }

    // キャラクターのグラフィックを描画する
    Game->Texture[TEX_MAN]->Draw(
        dx*sw, dy*sh, sw/3, sh/3, 0, 0, 1, 1, COL_BLACK);
}
```

キャラクターの進路を予測して表示する

「線路に沿って進むキャラクター」(→p. 147) について、これからキャラクターが進むルートを予測して表示する機能です。キャラクターの進路が簡単にわかるので、プレイヤーはより素早く線路の配置を変えることができ、快適なプレイが可能になります。これは「落下予測位置を表示する」(→p. 73) に似た機能です。

予測した進路は、通常の線路とは異なる色で表示されます (Fig. 3-19)。図では灰色で示しました。キャラクターが現在いる位置から、線路から落ちずに進める位置までが、灰色で表示されます。カーソルを操作して線路の配置を変えるたびに、進路の予測は変わります (Fig. 3-20)。キャラクターがどの位置で線路から落ちるのかが簡単にわかるので、プレイヤーはキャラクターが落ちる位置に注目しながら、効率的に線路を再配置することができます。

Fig. 3-19 予測した進路

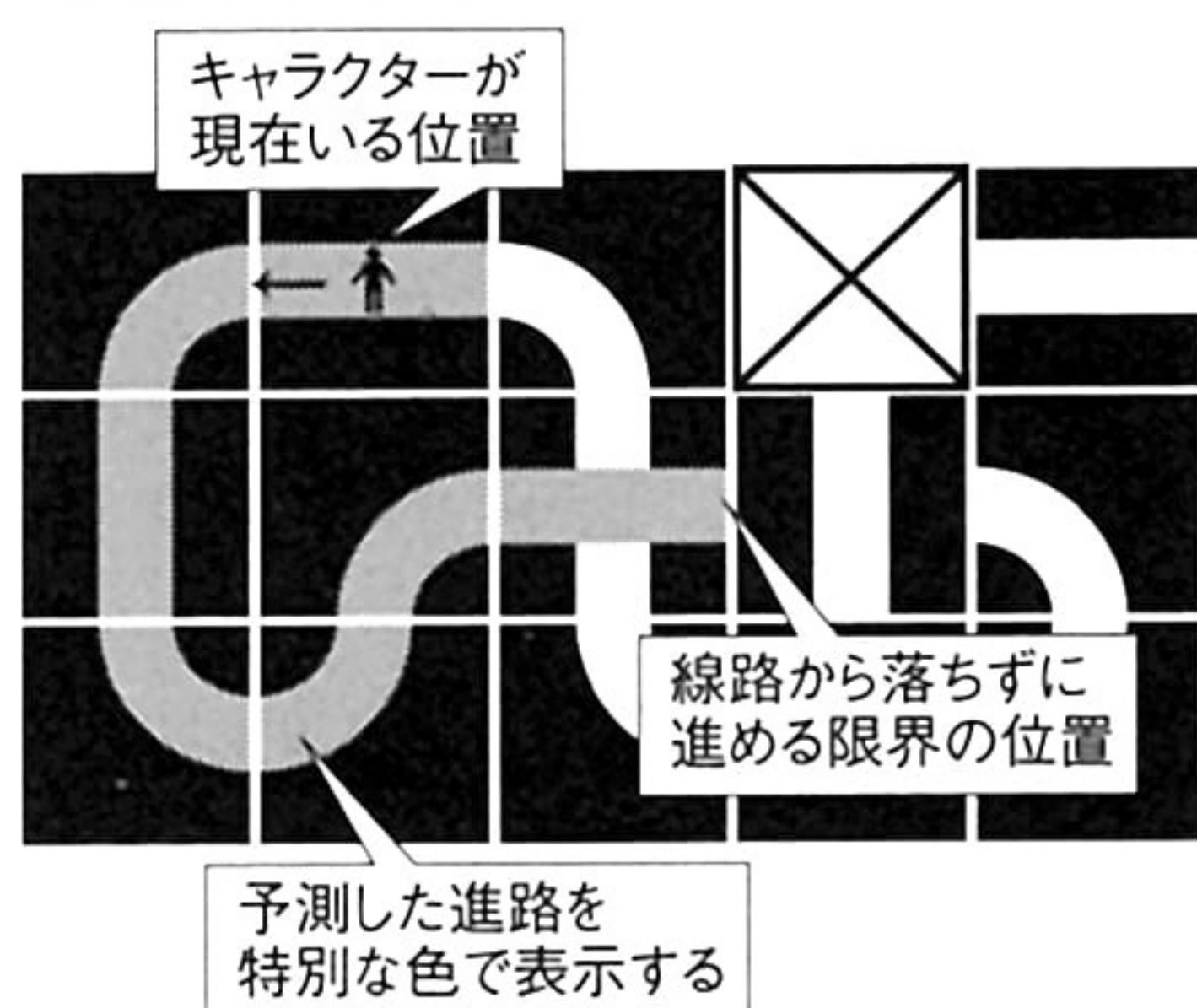
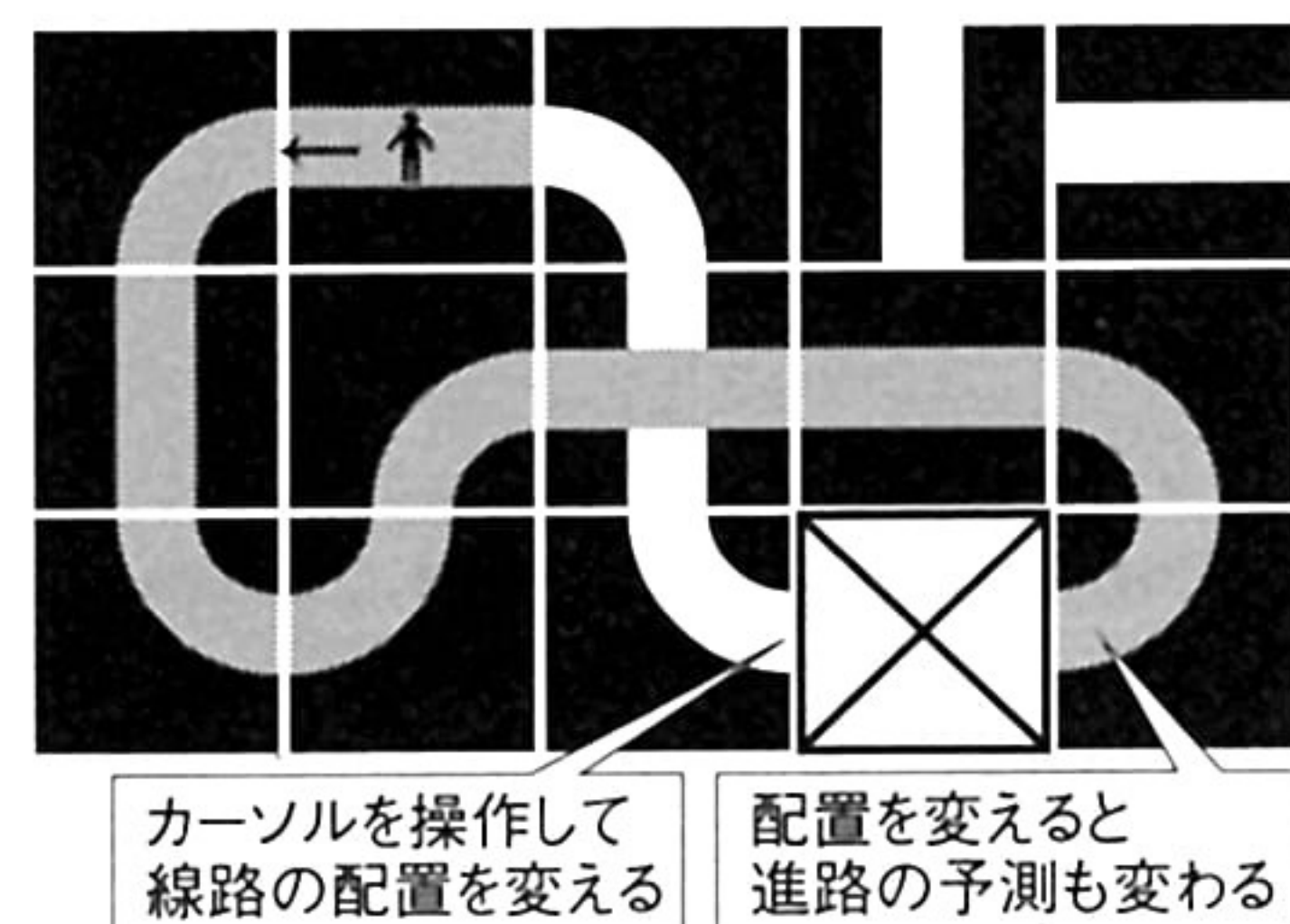



Fig. 3-20 進路の予測が変わる

アルゴリズム

キャラクターの進路を予測するには、キャラクターを線路に沿って進ませるときと同じ方法を使います。キャラクター自体は動かさずに、キャラクターの進路だけを予測して、その進路を通常の線路とは違った色で表示します。

予測した進路を記録するために、ステージのセルと同じサイズのセルを別に用意します (Fig. 3-21)。これを進路予測セルと呼ぶことにします。

進路予測セルには、そのセルにキャラクターが入るときの移動方向を記録します。同じセルを2回以上通るときには、例えば「右と上」のように、すべての移動方向を記録します。「1度目は右、2度目は上」のように、順番を記録する必要はありません。

移動方向を記録するときには、進路予測セルを調べて、以前に同じセルを同じ移動方向で通っていないかどうかを調べます (Fig. 3-22)。もしも同じセルを同じ移動方向で通ったことがあ

れば、その先の進路は予測ずみということなので、そこで進路の予測を終了します。

同じセルを同じ移動方向で通るという状況は、線路がループしているときに起こります。こういった場合には進路の予測を終了しないと、進路の予測が無限に終わらないことになってしまいます。

進路を予測する途中で、次の線路に入れなかったり、ステージの端から出てしまうこともあります。こういった場合にも、進路の予測を終了します（Fig. 3-23）。

予測した進路は、線路を描画するときに、一緒に描画します。いろいろな描画方法がありますが、本書のサンプルでは、最初に灰色の矩形を描き、その上に線路を重ねて描くことによって、進路を灰色で示しています（Fig. 3-24）。十字の線路の場合には、

・縦だけを通過する場合
・横だけを通過する場合
・縦と横の両方を通過する場合

の3種類に分けて描画します（Fig. 3-25）。

Fig. 3-21 進路予測セル

ステージ
キャラクターがセルに入るときの移動方向を進路予想セルに記録する
進路予測セル

左	左			
下	上	右		
下	右			

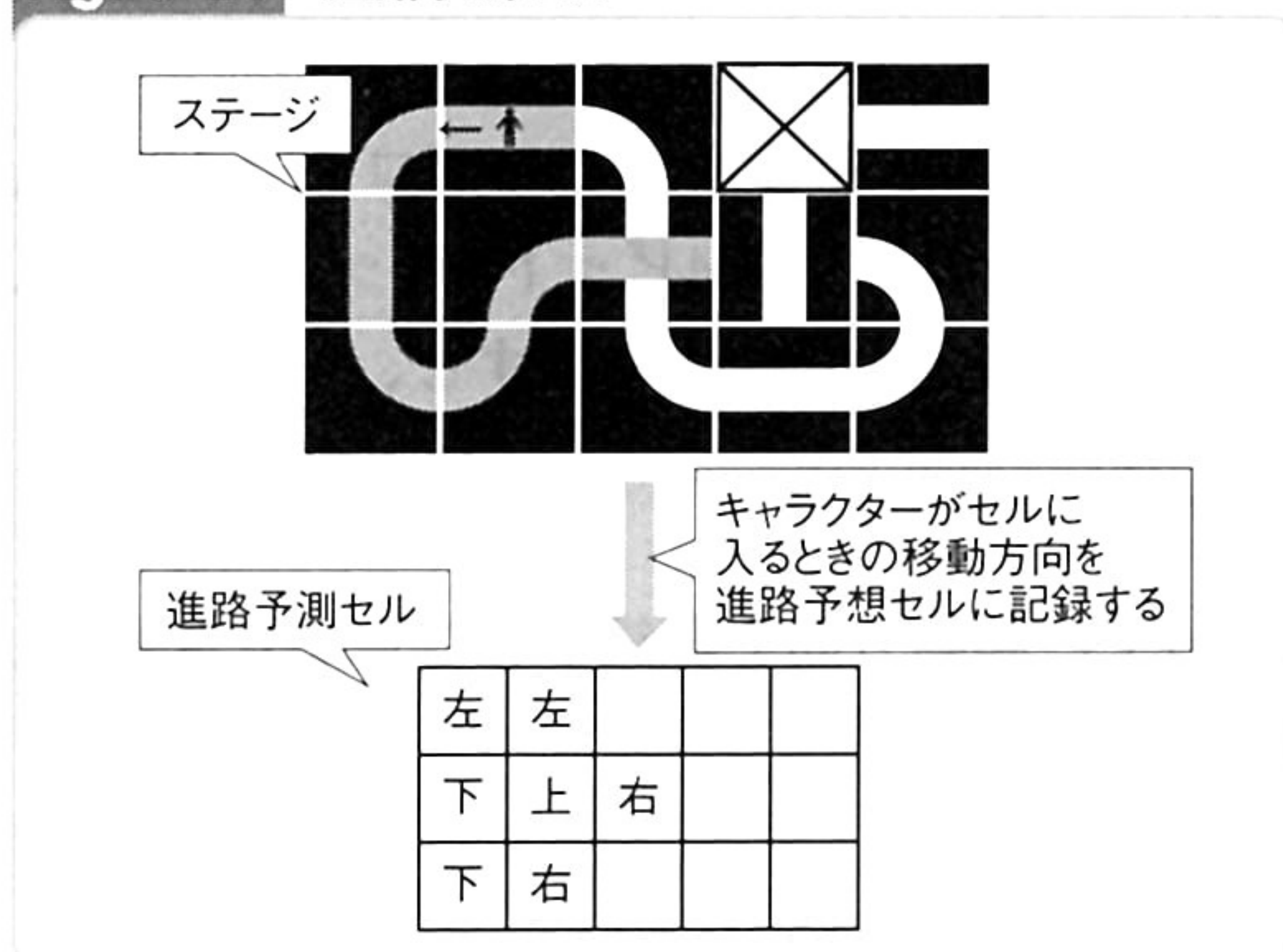



Fig. 3-22 線路がループしている場合に進路予測を終了する

このセルを同じ方向（左）に複数回通るので進路の予測を終了する
線路がループしている

左	左	上		
下	上	右と上	右	右
下	右	左	左	下

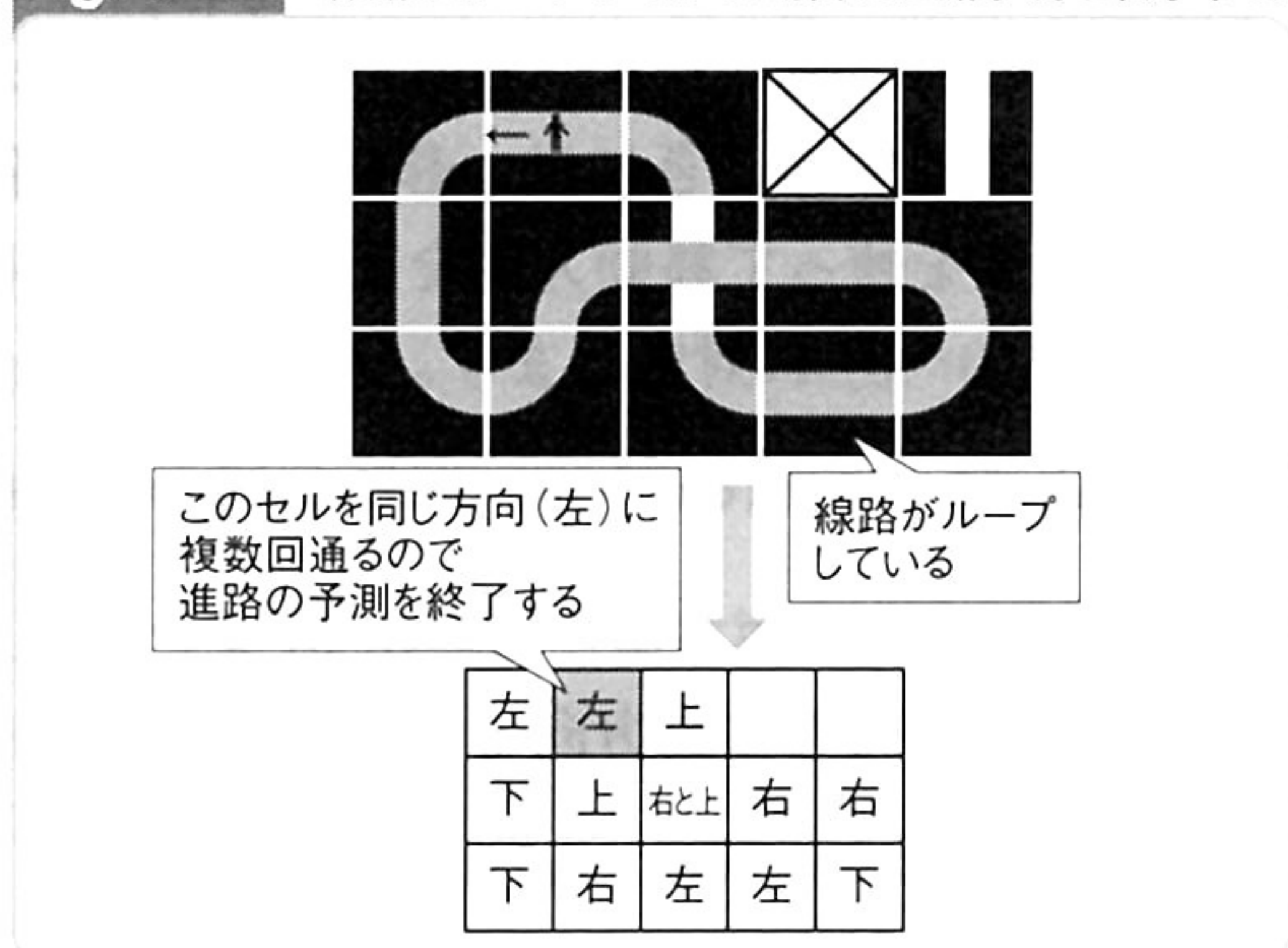



Fig. 3-23 行き止まりになった場合に進路予測を終了する

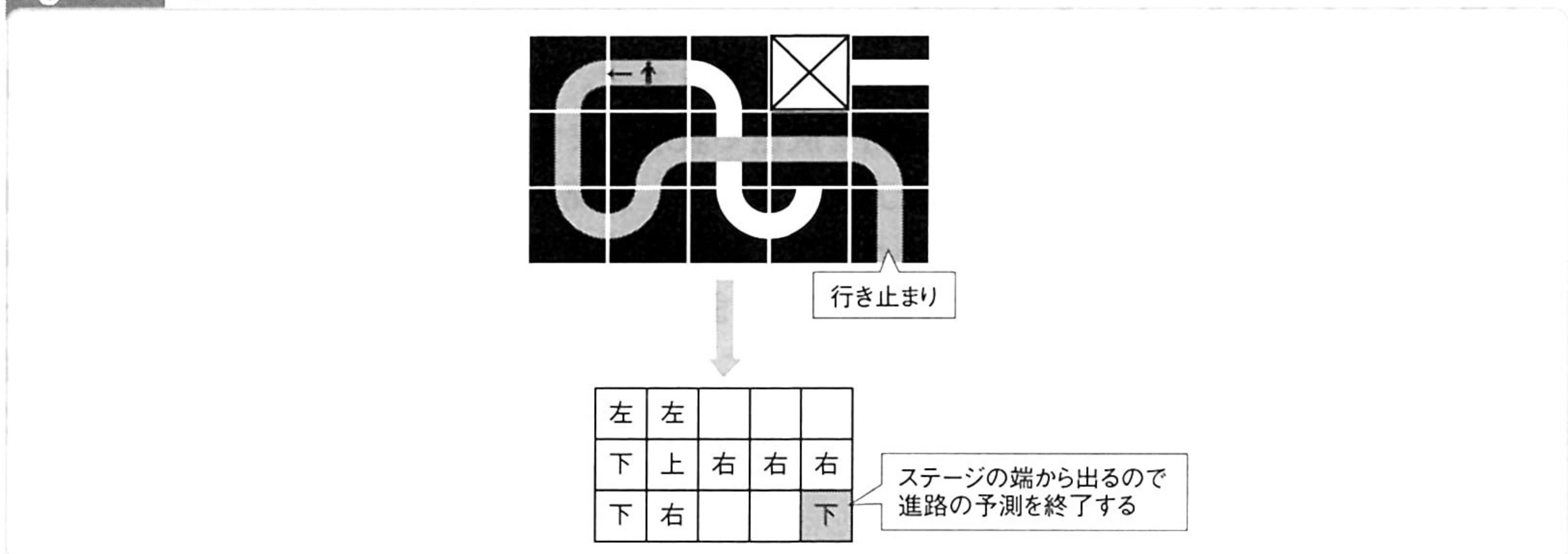
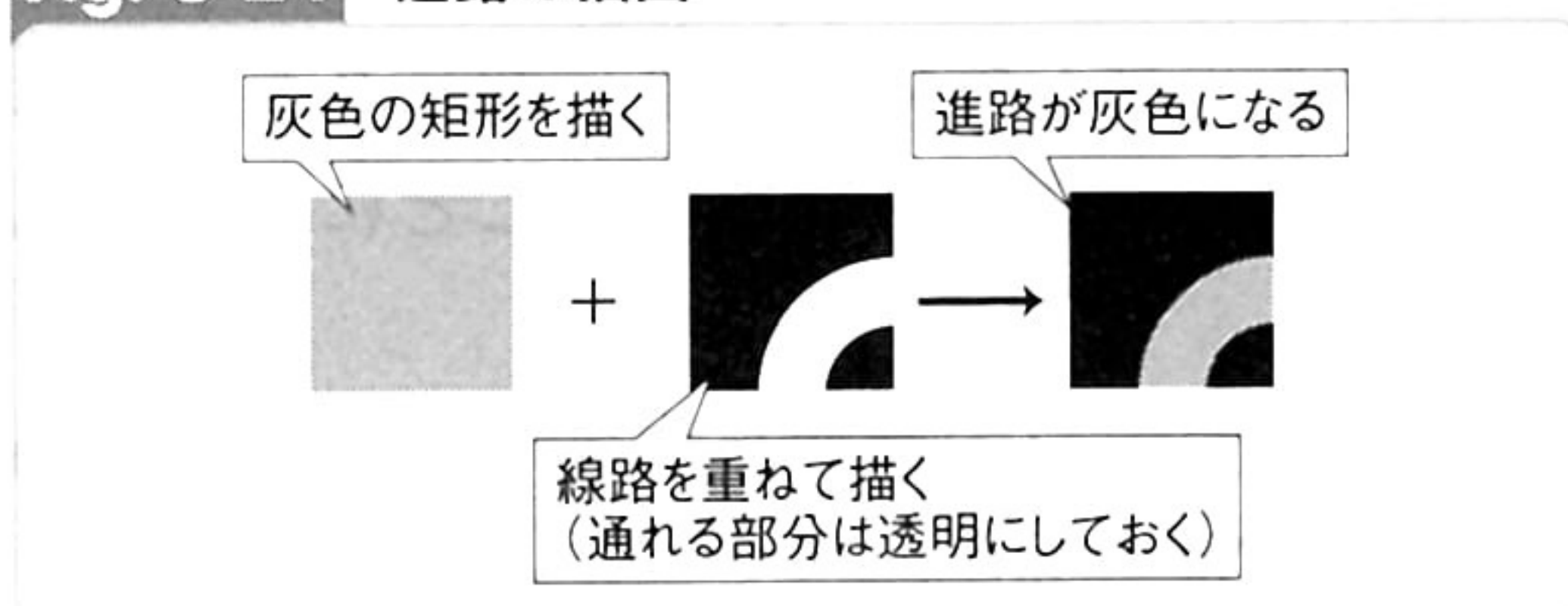
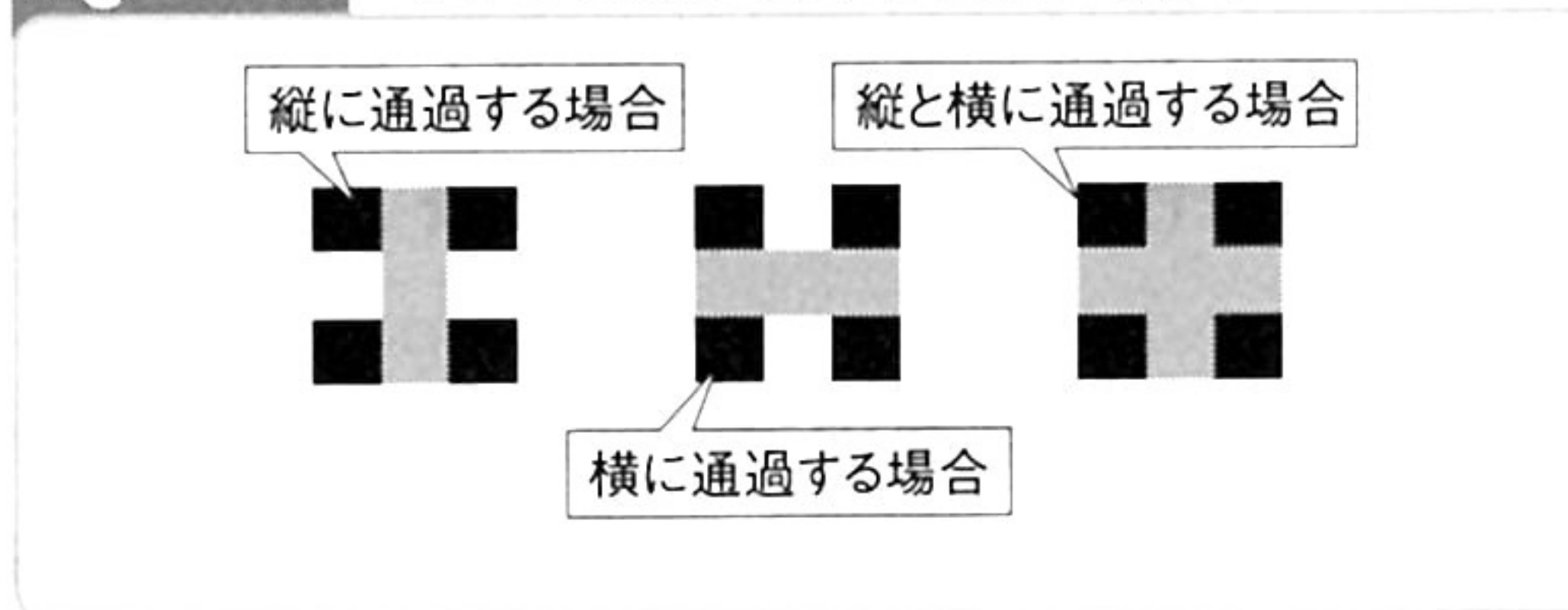
行き止まり
ステージの端から出るので進路の予測を終了する

左	左			
下	上	右	右	右
下	右			下

Fig. 3-24 進路の描画

灰色の矩形を描く
進路が灰色になる
線路を重ねて描く（通れる部分は透明にしておく）

Fig. 3-25 十字の線路に関する進路の描画

縦に通過する場合
縦と横に通過する場合
横に通過する場合

プログラム

List 3-3は進路を予測して表示するプログラムです。このサンプルでは、ステージの描画処理において、キャラクターの進路を予測して描画します。

最初に進路予測セル（PathCell）をクリアします。進路予測セルは、ステージのセルと同じサイズのセルです。

次に、キャラクターの現在の位置と移動方向を使って、進路の予測を開始します。キャラクターを動かすのと同じ方法で、線路に沿って進みます。同じ線路を同じ移動方向で通ったとき、ステージの外に出たとき、あるいは線路に入れないときには、予測を終了します。

セルを通過するときには、移動方向を進路予測セルに書き込みます。このプログラムでは、ビット演算を使って移動方向を記録することにしました。セルの下位4ビット（ビット0〜3）が、移動方向（0〜3）に対応します。座標を（cx, cy）、移動方向をdir（0〜3）で表すと、

```
PathCell->Set(cx, cy, PathCell->Get(cx, cy)|(1<<dir));
```

のように、セルの現在値と論理和（|）をとることによって、移動方向を記録することができます。

同じセルを同じ移動方向で通ったかどうかを調べるには、

```
PathCell->Get(cx, cy)&(1<<dir)
```

のように論理積（&）を使って、現在の移動方向（dir）に対応するビットが、セットされている（1になっている）かどうかを調べます。ビットがセットされていたら、同じ移動方向でそのセルを通ったことがある、ということです。

進路の予測が終わったら、線路に重ねて進路を描きます。進路のグラフィックを表示する前に、灰色の矩形を描くことによって、進路を灰色で表示します。

十字の線路に関しては、線路を通った方向によって、進路の表示方法を変えます。進路予測セルの値をpcとすると、「pc&0x05」という式で、線路を横に通ったかどうかがわかります。横に通ったときには、ビット0（右への移動を表す）またはビット2（左への移動を表す）が1になるので、この式の結果は0以外になります。同様に、「pc&0x0a」という式を使うと、線路を縦に通ったかどうかがわかります。縦に通ったときには、ビット1（下への移動を表す）またはビット3（上への移動を表す）が1になるので、この式の結果は0以外になります。

List 3-3 キャラクターの進路を予測して表示する(CConnectedRailStageクラス)

```
// ステージの描画処理
void CConnectedRailStage::Draw() {

    // 進路予測セル(進路予測用のセル)をクリアする
    PathCell->Clear(0);

    // キャラクターの現在の位置と移動方向を、
    // 進路予測の初期位置および初期方向とする
    int cx=Man->CX, cy=Man->CY, dir=Man->Dir;

    // 同じセルを同じ移動方向で通るまで、進路の予測を続ける
    while (!(PathCell->Get(cx, cy)&(1<<dir))) {

        // 現在のセルに現在の移動方向を書き込む
        PathCell->Set(cx, cy, PathCell->Get(cx, cy)|(1<<dir));

        // 次のセルに移動する
        cx+=CONNECTED_RAIL_VX[dir],
        cy+=CONNECTED_RAIL_VY[dir];

        // ステージ外に出たら、進路の予測を終了する
        if (
            cx<0 || cx>=Cell->GetXSize() ||
            cy<0 || cy>=Cell->GetYSize()
        ) break;

        // 線路の種類に応じて、移動方向を変化させる
        // 線路に入れない場合には、進路の予測を終了する
        int ndir=CONNECTED_RAIL_DIR[Cell->Get(cx, cy)-'0'][dir];
        if (ndir==4) break;
        dir=ndir;
    }

    // ...(中略)...

    // ステージのすべてのセルについて処理する
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {

            // ...(中略)...

            // 進路予測セルを調べる
            char pc=PathCell->Get(x, y);

            // セルが予測した進路になっている場合には、
            // 線路を描画する前に灰色の矩形を描画して、進路であることを示す
            if (pc) {
```

```
                    // 十字の線路を縦（上から下、または下から上）だけに
                    // キャラクターが通る場合の描画
                    if (c==7 && !(pc&0x05)) {
                        Game->Texture[TEX_FILL]->Draw(
                            dx+sw/3, dy, sw/3, sh,
                            0, 0, 1, 1, COL_LGRAY);
                    } else

                    // 十字の線路を横（左から右、または右から左）だけに
                    // キャラクターが通る場合の描画
                    if (c==7 && !(pc&0x0a)) {
                        Game->Texture[TEX_FILL]->Draw(
                            dx, dy+sh/3, sw, sh/3,
                            0, 0, 1, 1, COL_LGRAY);
                    } else

                    // キャラクターが、曲がった線路・まっすぐな線路を通る場合、
                    // または十字の線路を縦と横の両方向に通る場合の描画
                    {
                        Game->Texture[TEX_FILL]->Draw(
                            dx, dy, sw, sh,
                            0, 0, 1, 1, COL_LGRAY);
                    }
                }

                // 灰色の進路に重ねて、線路のグラフィックを描画する
                Game->Texture[TEX_RAIL0+c]->Draw(
                    dx, dy, sw, sh,
                    0, 0, 1, 1, COL_BLACK);
            }
        }

        // ...（中略）...
    }
```

パイプをつなぐ

パイプをつないでルートを作るアクションです。パイプをつなぐと、パイプを通して水や油といった液体を運ぶことができます。パイプを上手につないで、目的の場所まで液体をこぼさずに運ぶことが、ゲームの目的です。

「線路をつなぐ」（→p. 140）と同様に、パイプにはいろいろな形があります（Fig. 3-26）。詳細はゲームによって違いますが、曲がったパイプ、まっすぐなパイプ、十字のパイプなどが代表的です。ステージにはパイプを格子状に並べることができます（Fig. 3-27）。パイプ同士がつな

がるように、パイプの種類や位置を上手に選ぶ必要があります。

パイプを並べる方法はゲームによって異なります。パイプを好きな位置に置けるゲームもあれば、パイプが画面の上方から降ってくるゲームもあります。本書のサンプルでは、カーソルをレバー入力で上下左右に動かして、好きな位置にパイプを置けるようにしました(Fig. 3-28)。配置するパイプの形状は、ランダムに決まります。

ステージには、液体が流れ出る水源やポンプなども配置されています。水源にパイプを接続すると、パイプを通して液体を運ぶことができます(Fig. 3-29)。本書のサンプルでは、液体の進路を灰色で表示することにしました。

パイプをつなぐゲームには『パイプドリーム』などがあります。このゲームの目的は、スタートからゴールまでパイプをつなげて、液体を運ぶことです。カーソルを動かして、ステージ内の任意の位置にパイプを置くことができます。開始から一定時間が経過すると、スタートから液体が流れ出します。液体がパイプからこぼれる前に、素早くパイプをゴールまでつながなければなりません。使うパイプの本数の下限が決められているので、パイプを上手に使って、できるだけ長いルートを作る必要があります。

パイプを使った別のゲームには『カチャット』があります。このゲームでは、ステージ上方から次々にパイプが降ってきます。カーソルを合わせてボタンを押すと、任意のパイプを回転させることができます。パイプを回転させてルートを作り、画面の左右に並んでいる水源同士をつなぐと、パイプを消すことができます。ステージが埋まってしまわないように、次々にパイプを消すことがゲームの目的です。

その他、『線脳』もパイプをつなぐゲームです。『カチャット』に似ていますが、このゲーム

Fig. 3-26 いろいろなパイプ

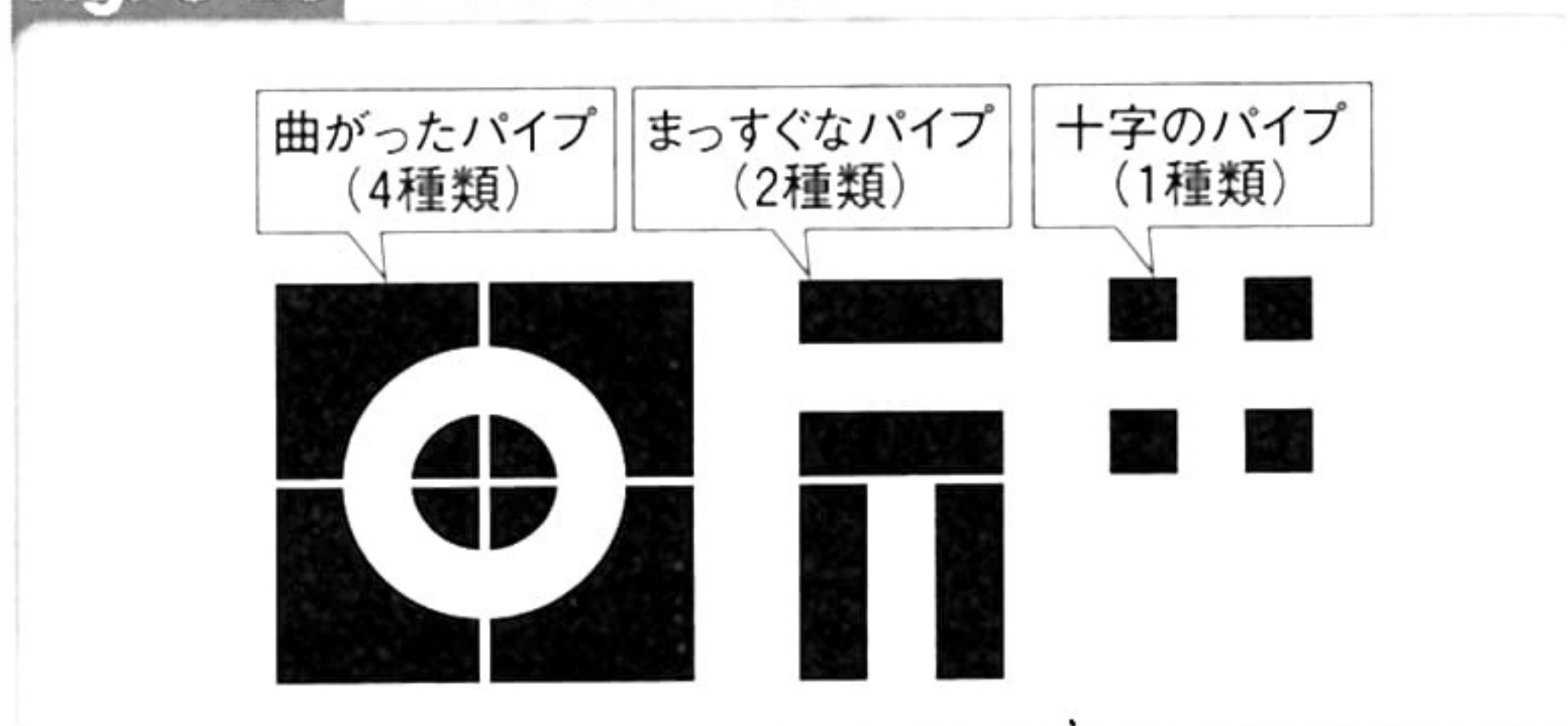



Fig. 3-27 パイプを並べる

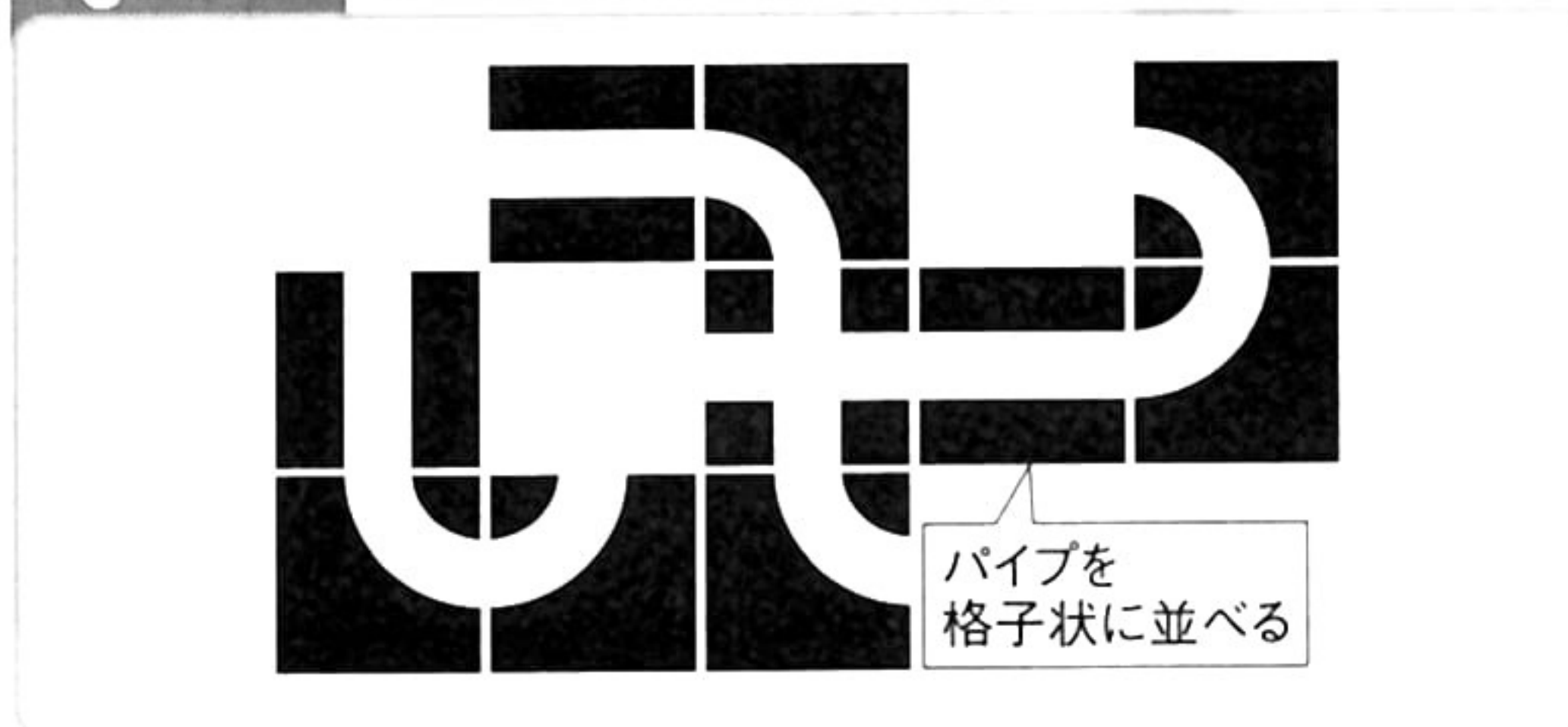



Fig. 3-28 パイプを好きな位置に置く

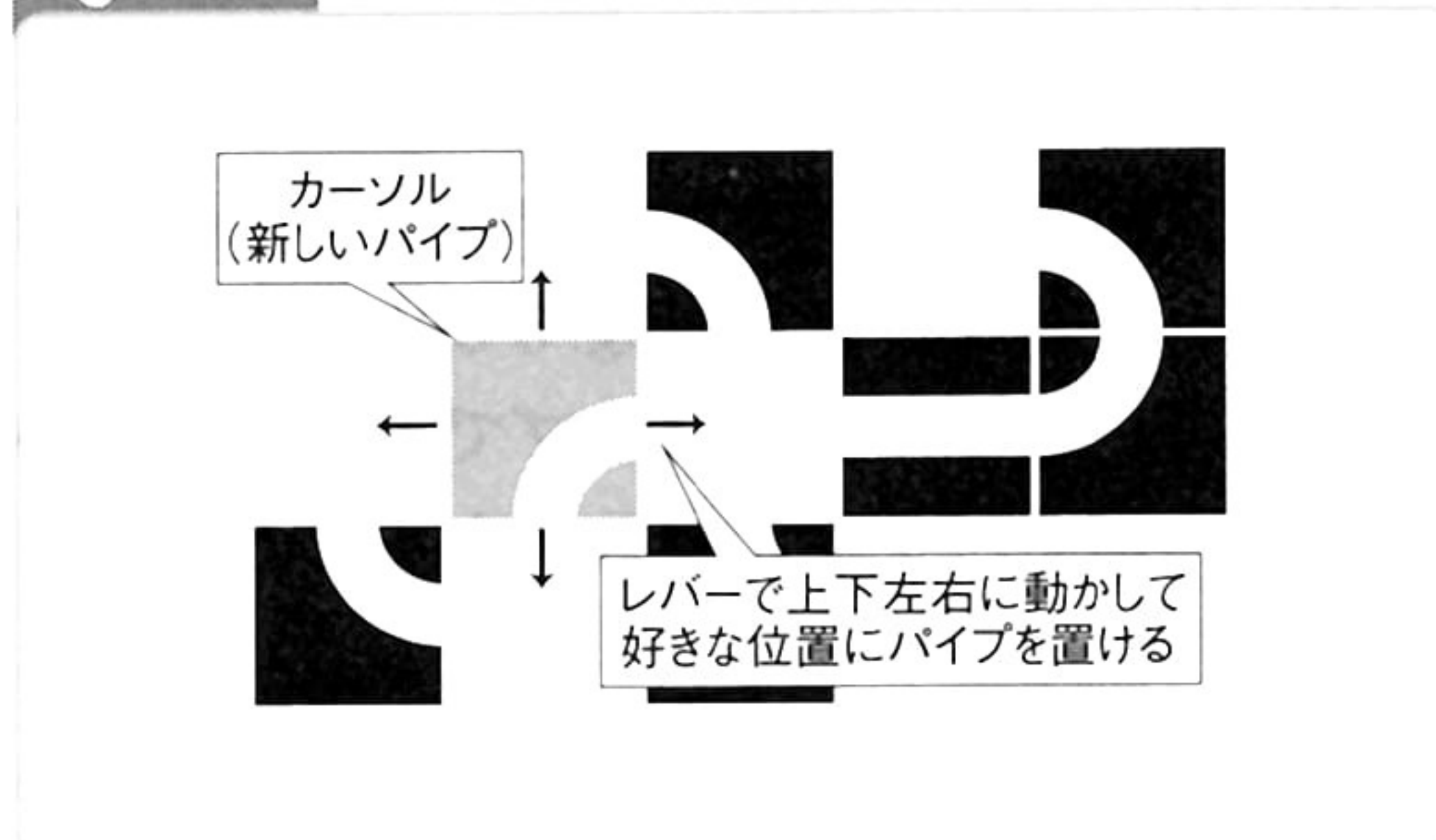
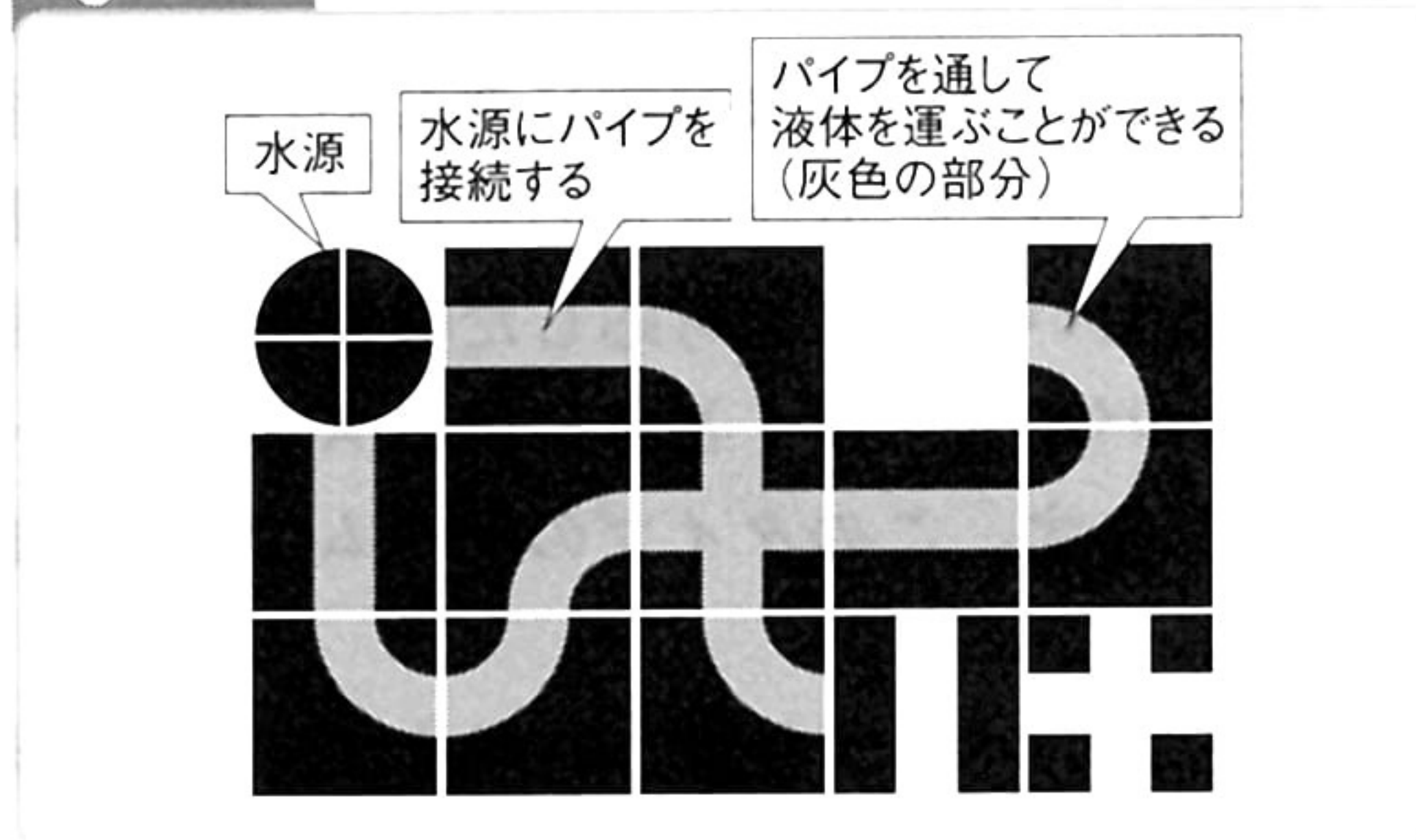



Fig. 3-29 水源からパイプで液体を運ぶ

では落下中のパイプを移動したり回転させたりすることができます。パイプをつないでループを作ると消えます。

アルゴリズム

パイプをつなぐアクションは、「線路をつなぐ」(→p. 140)と似た方法で実現することができます。線路の場合と同様に、パイプやステージをセルで表します。

まず、パイプをセルで表現します(Fig. 3-30)。パイプにはいろいろな形があるので、形状ごとに異なる数字を割り振ります。ここでは7種類の形に対して、数字の「1〜7」を割り振ることにしました。

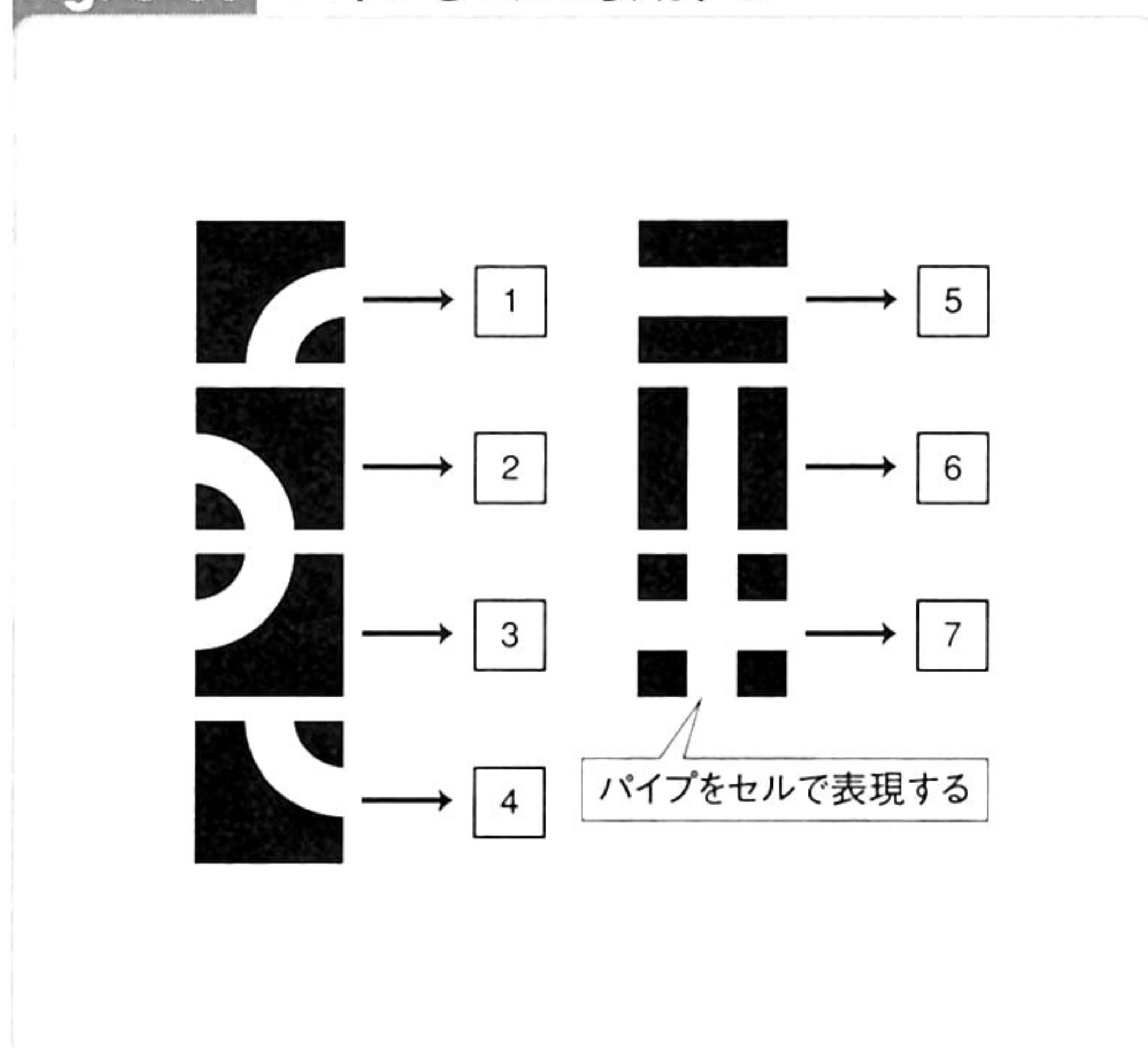
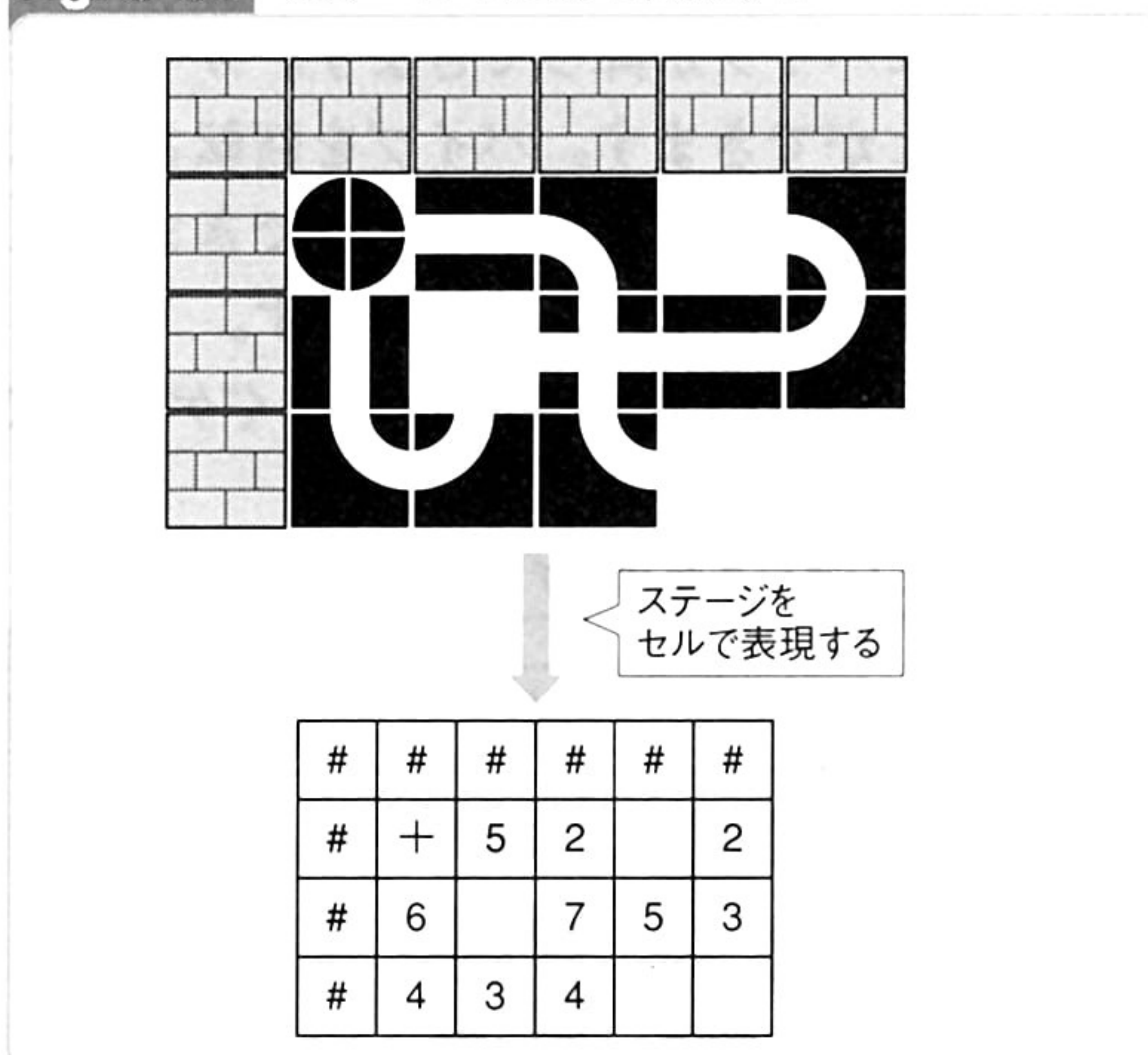
次に、ステージをセルで表現します(Fig. 3-31)。ステージにはパイプのほかに、水源や壁もあります。水源は「+」で、壁は「#」で表すことにしました。

Fig. 3-30 パイプをセルで表現する

1
2
3
4
5
6
7
パイプをセルで表現する

Fig. 3-31 ステージをセルで表現する

ステージを
セルで表現する

#	#	#	#	#	#
#	+	5	2		2
#	6		7	5	3
#	4	3	4		

パイプを配置する

プレイヤーは好きな位置に新しいパイプを置くことができます。カーソルを動かして、ボタンを入力すると、カーソルの位置にパイプが配置されます。これは、カーソルの位置にあるセルに、新しいパイプに対応した数字を書き込むことによって実現します(Fig. 3-32)。

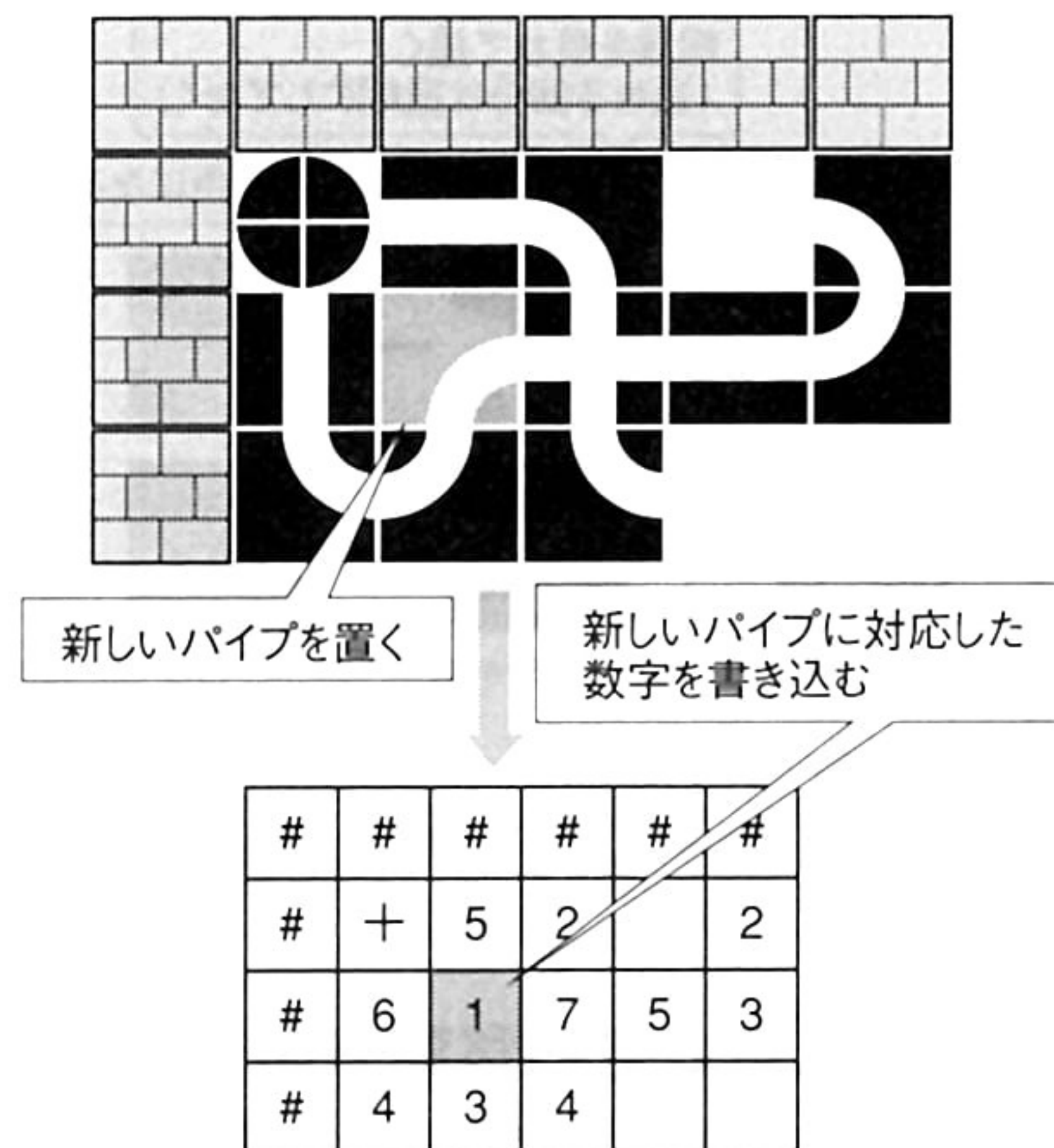
新しいパイプを置くときには、ステージのセルを調べて、水源や壁をパイプで上書きしないようにします。このタイプのゲームでは一般的に、水源や壁をパイプに置き換えることはできません。

一方、古いパイプを新しいパイプに置き換えられるかどうかは、ゲームによって異なります。

Fig. 3-32 新しいパイプを置く

新しいパイプを置く

新しいパイプに対応した数字を書き込む

#	#	#	#	#	#
#	＋	5	2		2
#	6	1	7	5	3
#	4	3	4		

本書のサンプルでは自由に置き換えられることにしました。ゲームによっては、パイプを置き換えるとスコアやタイムが減る、といったペナルティを課しているものもあります。

液体の進路を予想する

水源にパイプがつながったら、パイプを液体が通過する様子を表示します。これは「キャラクターの進路を予測して表示する」(→p. 154) と同じ方法で実現できます。水源を開始位置として、キャラクターが線路に沿って進む場合と同じように、パイプに沿って進む液体の進路を予測します。

Fig. 3-33 進路予測セル

ステージ

液体がセルに入るときの移動方向を進路予測セルに記録する

進路予測セル

		右	右		上
	下	上	右と下	右	右
	下	右	下		

Fig. 3-34 進路予測の終了

水源から出たパイプが別の方向から水源に戻っている

水源にぶつかるので進路の予測を終了する

		右	右		
	上	左	下と左	左	上
	左	下	下	右	右

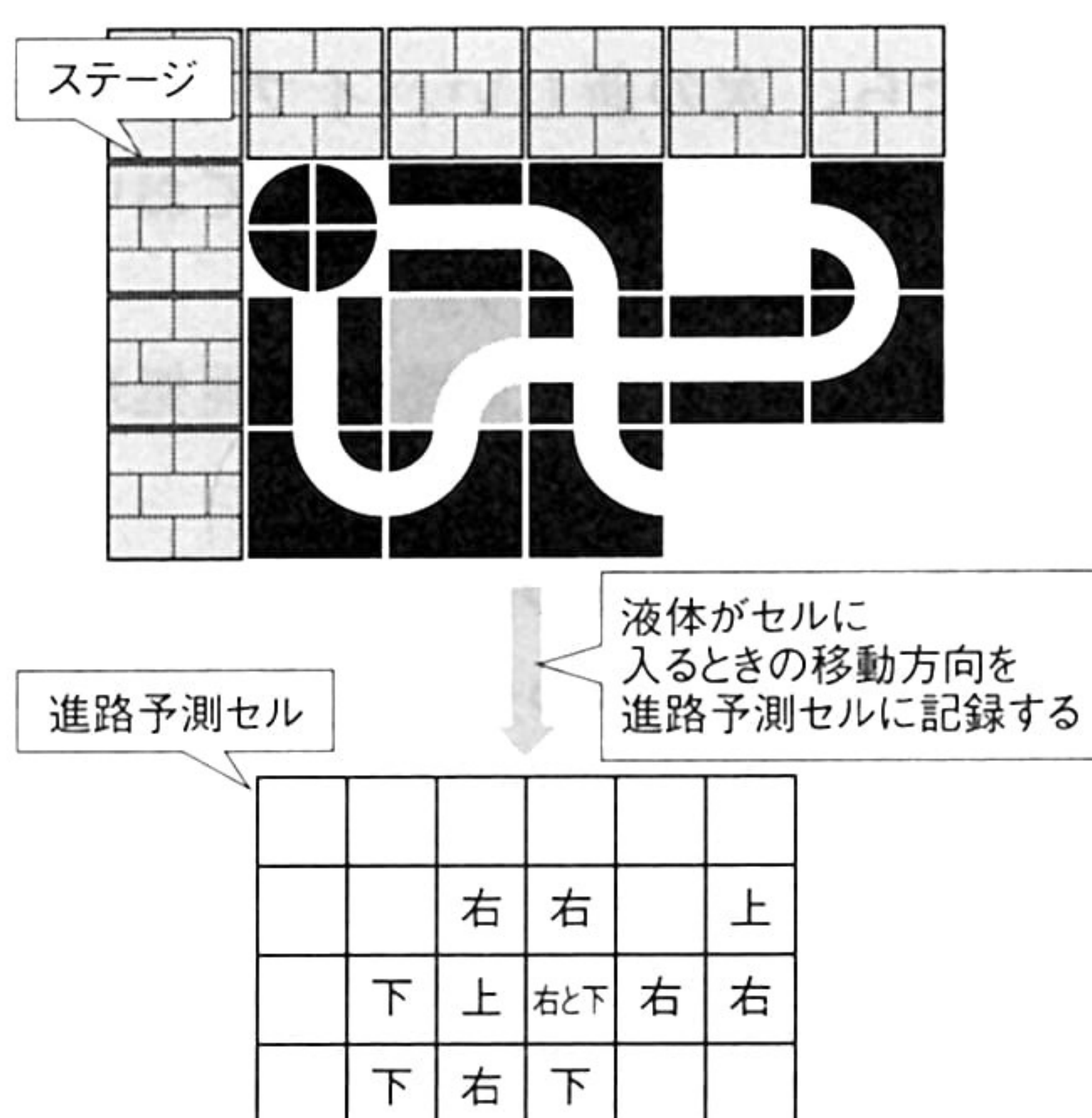
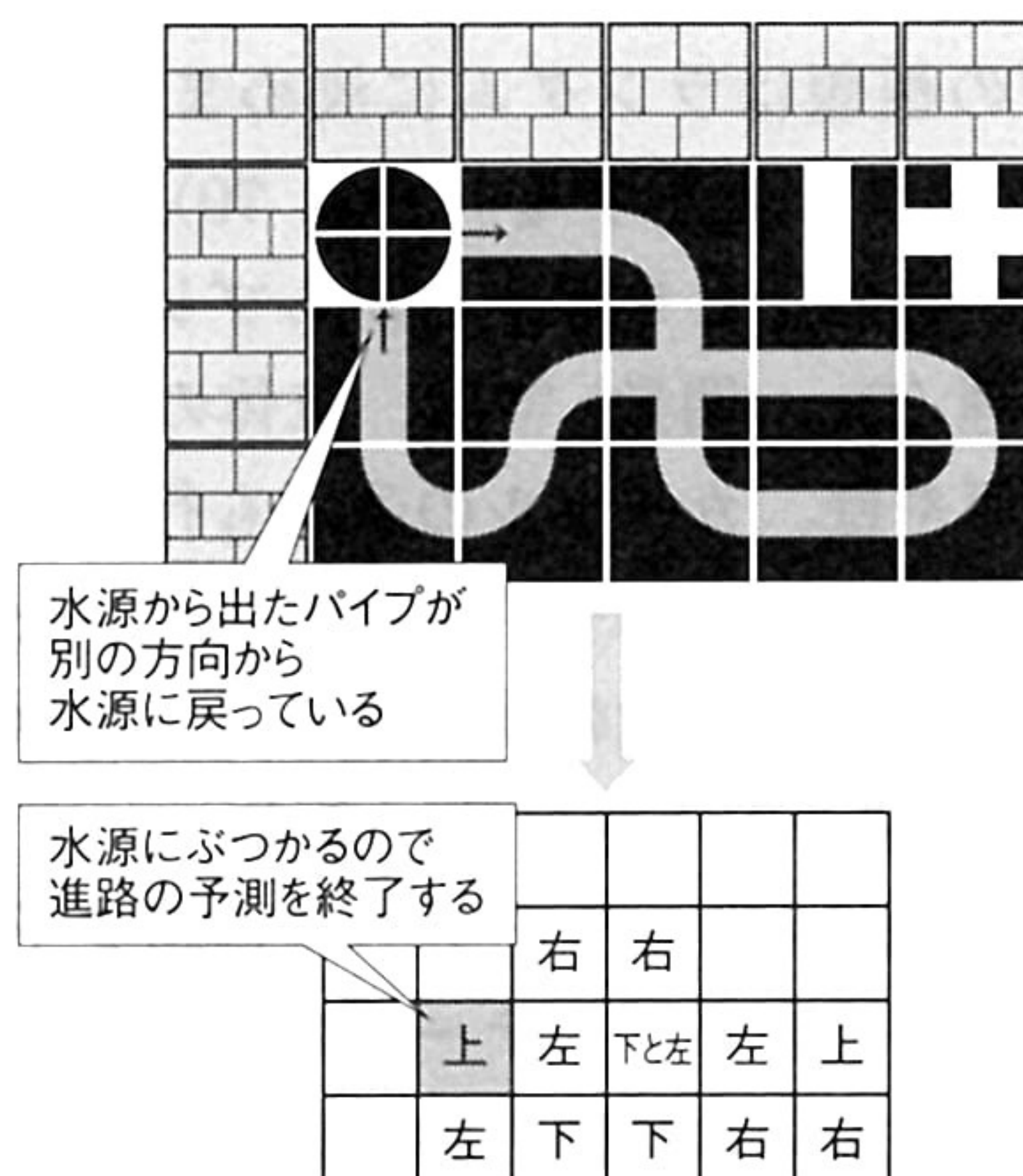
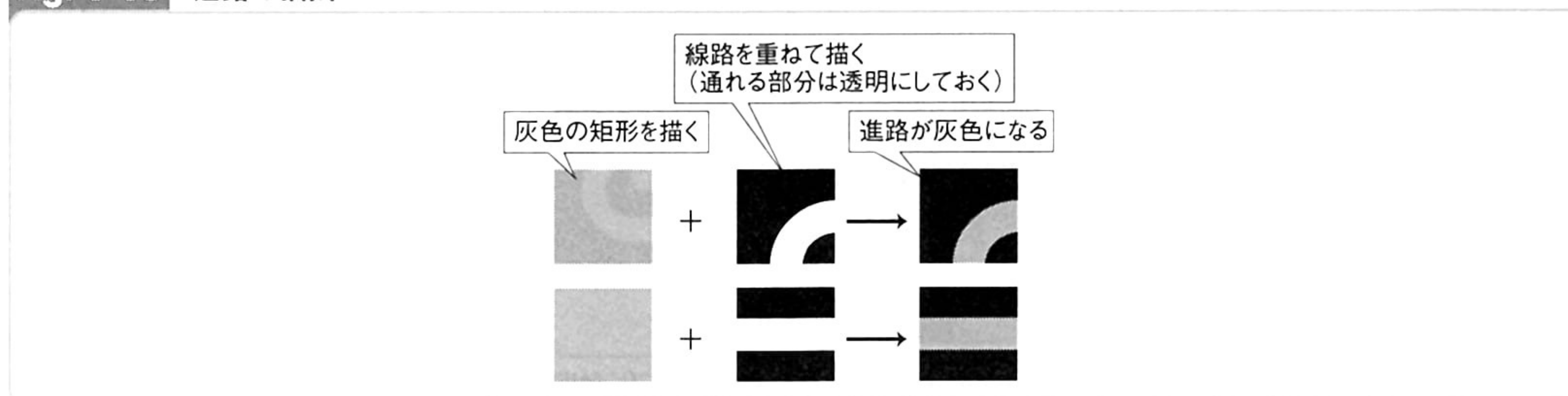



Fig. 3-35 進路の描画

予測した進路を記録するために、ステージのセルと同じサイズのセルを用意します。これを進路予測セルと呼びます（Fig. 3-33）。

進路予測セルには、セルに液体が入るときの移動方向を記録します。例えば、液体が右に向かって入るときには、「右」と記録します。また、右に向かって入るときと下に向かって入るときの両方がある場合には、「右と下」のように、すべての移動方向を記録します。進路を予測している途中で、壁や水源にぶつかったら、予測を終了します（Fig. 3-34）。

予測した進路は、パイプと一緒に描画します。本書のサンプルでは、灰色の矩形をパイプの下に重ねて描くことによって、進路を灰色で示します（Fig. 3-35）。

プログラム

List 3-4はパイプをつなぐプログラムです。ステージの移動処理と描画処理を掲載しました。

移動処理（Move関数）では、カーソルを動かす処理と、新しいパイプを配置する処理を行います。レバー入力でカーソルを上下左右に動かし、ボタン入力があったら、カーソルの位置に新しいパイプを置きます。このサンプルでは、カーソルが水源や壁の上には入れないようにして、水源や壁をパイプで置き換えるのを防いでいます。

パイプの種類はランダムに決めます。パイプを置いたら、次の新しいパイプを決めます。「次のブロックを表示する」（→p. 70）のように、次のパイプをあらかじめ表示しておいて、プレイヤーが先のことを考えながらプレイできるようにしてもよいでしょう。

描画処理（Draw関数）では、液体の進路を予測する処理と、パイプや進路の描画処理を行います。水源や壁、カーソルの表示も行います。

最初は進路の予測です。進路予測セルをクリアしておき、水源からの液体の進路を予測します。ステージには複数の水源があり、各水源の上下左右に液体が流れるので、すべての水源について4通りずつの予測を行います。

進路の予測は「キャラクターの進路を予測して表示する」（→p. 154）とほぼ同じプログラムです。線路の場合と同じテーブルを使い、パイプの形に沿って液体の進路を予測します。進路予測セルには液体の移動方向を書き込みます。水源や壁にぶつかったら、予測を終了します。

進路を予測したら、パイプや進路を描画します。進路を描くには、先に灰色の矩形を描き、これに重ねてパイプのグラフィックを描きます。十字のパイプについては、縦に通る場合・横に通る場合・縦横に通る場合、の3種類について、それぞれ異なる表示を行います。

List 3-4　パイプをつなぐ(CConnectedPipeStageクラス)

```
// ステージの移動処理
bool CConnectedPipeStage::Move(const CInputState* is) {

    // レバー入力に応じてカーソルを移動させる
    int cx=CX, cy=CY;
    if (!PrevLever) {

        // レバーの入力方向に応じて、新しい座標を求める
        if (is->Left) cx--; else
        if (is->Right) cx++; else
        if (is->Up) cy--; else
        if (is->Down) cy++;

        // 新しい座標のセルが壁や水源でなければ、
        // カーソルを新しい座標に移動させる
        char c=Cell->Get(cx, cy);
        if (c!='#' && c!='+') {
            CX=cx;
            CY=cy;
        }
    }

    // 前回のレバー入力を記録する
    PrevLever=is->Left|is->Right|is->Up|is->Down;

    // ボタン入力があった場合には、
    // カーソルの位置に新しいパイプを置く
    if (!PrevButton && is->Button[0]) {

        // セルに新しいパイプの種類を書き込む
        Cell->Set(cx, cy, '0'+NewPipe);

        // 次のパイプの種類をランダムに決める
        NewPipe=Rand.Int31()%CONNECTED_PIPE_COUNT+1;
    }

    // 前回のボタン入力を記録する
    PrevButton=is->Button[0];

    return true;
}

// ステージの描画処理
```

```
void CConnectedPipeStage::Draw() {

    // セルのサイズ
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 進路予測セルをクリアする
    PathCell->Clear(0);

    // ステージ内にあるすべての水源について、
    // 水源からパイプを伝って水が流れる進路を予測する
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {

            // 水源のセルを探す
            if (Cell->Get(x, y)=='+') {

                // すべての方向(右・下・左・上)について、
                // 進路を予測する
                for (int i=0; i<4; i++) {

                    // 初期の位置と移動方向を設定する
                    int cx=x, cy=y, dir=i;

                    // 同じセルを同じ移動方向で通るまで、
                    // 進路の予測を続ける
                    while (!(PathCell->Get(cx, cy)&(1<<dir))) {

                        // 現在のセルに現在の移動方向を書き込む
                        PathCell->Set(
                            cx, cy, PathCell->Get(cx, cy)|(1<<dir));

                        // 次のセルに移動する
                        cx+=CONNECTED_RAIL_VX[dir],
                        cy+=CONNECTED_RAIL_VY[dir];
                        char c=Cell->Get(cx, cy)-'0';

                        // パイプ以外のセル(壁・水源・空のセル)に
                        // ぶつかった場合には、進路の予測を終了する
                        if (
                            c<0 ||
                            c>=CONNECTED_PIPE_COUNT+1
                        ) break;

                        // パイプの種類に応じて、移動方向を変化させる
                        // パイプに入れない場合には、進路の予測を終了する
                        int ndir=CONNECTED_RAIL_DIR[c][dir];
                        if (ndir==4) break;
                        dir=ndir;
                    }
```

```
                }
            }
        }
    }

    // 描画サイズの計算
    float
        sw=Game->GetGraphics()->GetWidth()/xs,
        sh=Game->GetGraphics()->GetHeight()/ys;

    // ステージのすべてのセルを描画する
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {

            // セルの種類を取得する
            char c=Cell->Get(x, y);

            // セルの種類に応じてグラフィック（水源・壁）を選ぶ
            int t=TEX_VOID;
            if (c=='+') {
                t=TEX_CONVEYOR1;
            } else
            if (c=='#') {
                t=TEX_FLOOR;
            } else

            // セルがパイプの場合
            if ('0'<=c && c<'0'+CONNECTED_PIPE_COUNT+1) {

                // パイプの形状に対応したグラフィックを選ぶ
                t=TEX_RAIL0+c-'0';

                // 進路予測セルを調べる
                char pc=PathCell->Get(x, y);

                // セルが予測した進路になっている場合には、
                // パイプを描画する前に灰色の矩形を描画して、
                // 進路であることを示す
                if (pc) {

                    // 十字のパイプを縦（上から下、または下から上）
                    // だけに進路が通る場合
                    if (c=='7' && !(pc&0x05)) {
                        Game->Texture[TEX_FILL]->Draw(
                            x*sw+sw/3, y*sh, sw/3, sh,
                            0, 0, 1, 1, COL_LGRAY);
                    } else

                    // 十字のパイプを横（左から右、または右から左）
```

```
                    // だけに進路が通る場合
                    if (c=='7' && !(pc&0x0a)) {
                        Game->Texture[TEX_FILL]->Draw(
                            x*sw, y*sh+sh/3, sw, sh/3,
                            0, 0, 1, 1, COL_LGRAY);
                    } else

                    // 曲がったパイプ・まっすぐなパイプを通る場合、
                    // または十字のパイプを縦と横の両方向に通る場合
                    {
                        Game->Texture[TEX_FILL]->Draw(
                            x*sw, y*sh, sw, sh,
                            0, 0, 1, 1, COL_LGRAY);
                    }
                }
            }

            // パイプを描画する
            Game->Texture[t]->Draw(
                x*sw, y*sh, sw, sh, 0, 0, 1, 1, COL_BLACK);
        }
    }

    // カーソル(新しいパイプ)を描画する
    Game->Texture[TEX_RAIL0+NewPipe]->Draw(
        CX*sw, CY*sh, sw, sh, 0, 0, 1, 1, COL_LGRAY);
}
```

SAMPLE

「CONNECTED PIPE」は「パイプをつなぐ」のサンプルです。レバーの上下左右(カーソルキーの上下左右)でカーソルが移動します。灰色で表示されているパイプがカーソルです。

ボタン0(Zキー)を押すと、新しいパイプをカーソルの位置に置くことができます。水源や壁の上に置くことはできませんが、すでに置かれたパイプを置き換えることはできます。水源にパイプをつなぐと、液体の進路が灰色で表示されます。

CONNECTED PIPE → p. 387

結合して形を作る

浮遊する物体を結合して、目的の形を作るというアクションです。狙った位置で物体を結合して、大きく複雑な形を作り上げることが、このアクションの面白さです。

自機はレバー入力で上下左右に動かすことができます。ステージには自機の他に、複数の物

体が浮遊しています（Fig. 3-36）。

自機を物体に接触させると、物体と結合することができます（Fig. 3-37）。結合する位置は、自機の左上・左下・右上・右下のいずれかで、物体にどの方向から接触したかによって決まります。結合した物体は、自機とともに動きます。

物体と結合した状態で、自機または物体を別の物体に接触させると、さらに結合することができます（Fig. 3-38）。これを次々に繰り返すと、多数の物体と結合して、複雑な形を作ることができます（Fig. 3-39）。

物体の数が増えるにつれて、思ったとおりの場所に物体を結合しにくくなることが、このアクションの面白いところです。思った場所に物体を結合できなかったら、ボタンを入力すれば、最後に結合した物体を切り離すことができます（Fig. 3-40）。連続してボタンを入力すると、物体を結合したのとは逆の順序で次々に分離していき、最後は自機だけの状態に戻ります。

結合して形を作るアクションを採用したゲームには『フォゾン』があります。このゲームでは、自機を操作して物体と結合し、指定された形を作ることが目的です。敵に自機が接触する

Fig. 3-36 自機と物体

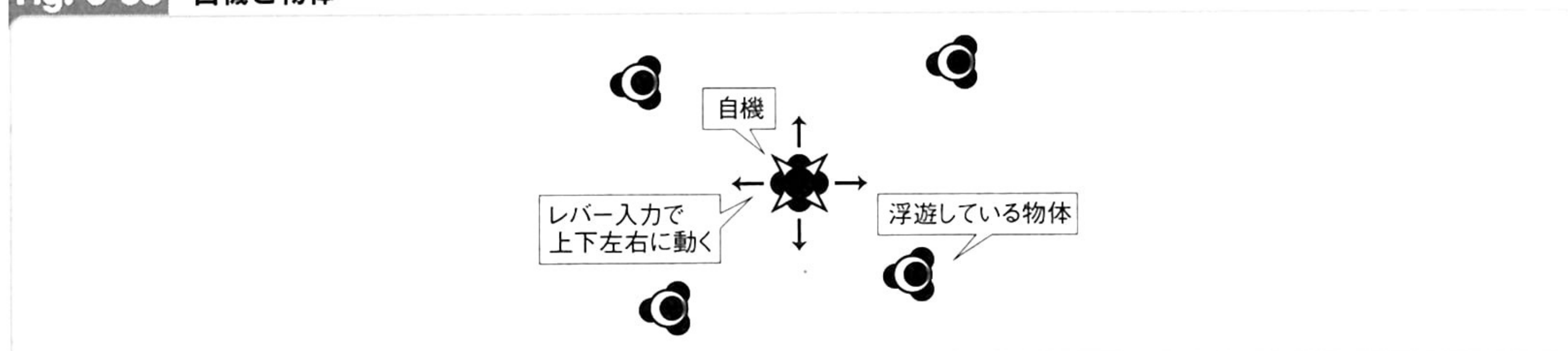



Fig. 3-37 物体と結合する

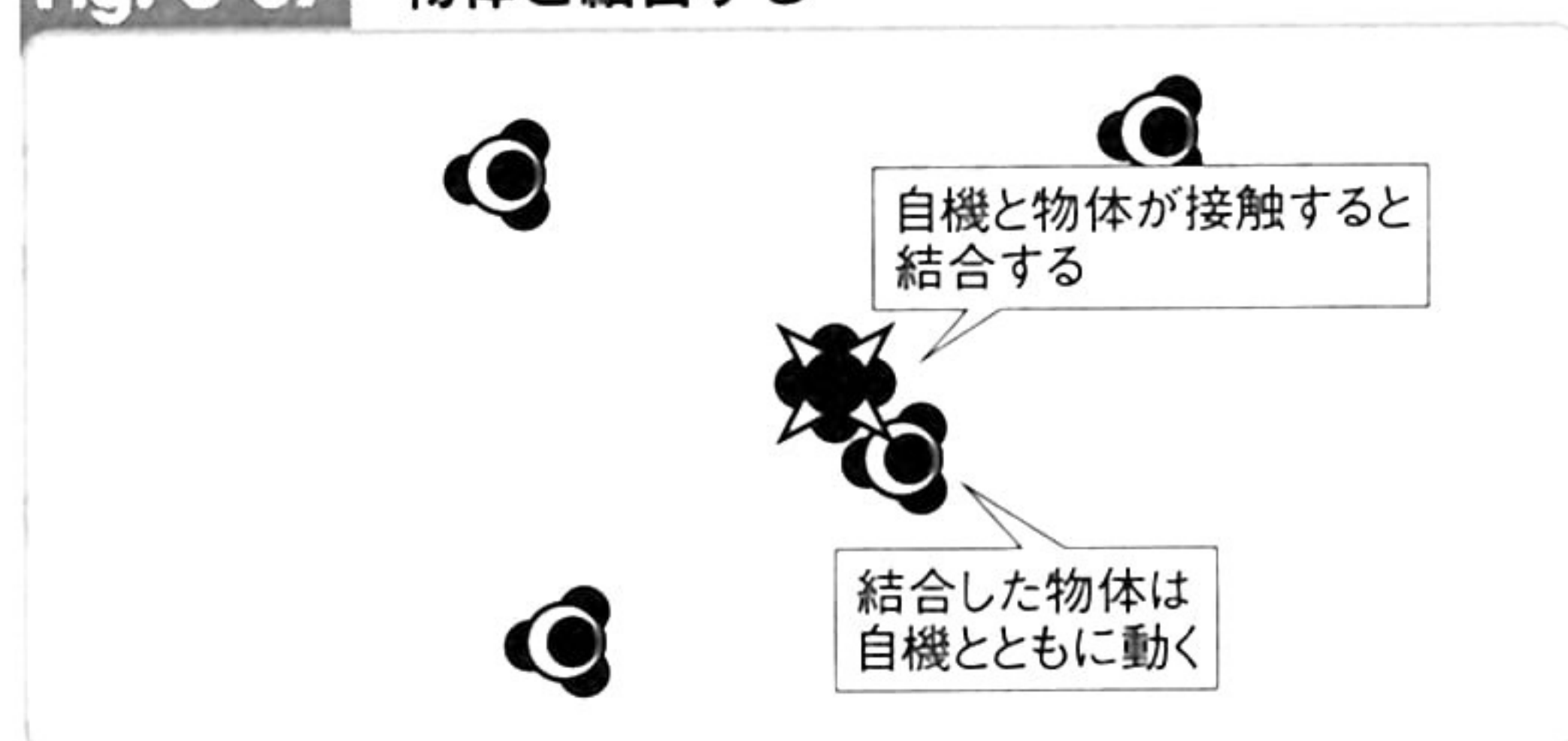



Fig. 3-38 別の物体と結合する

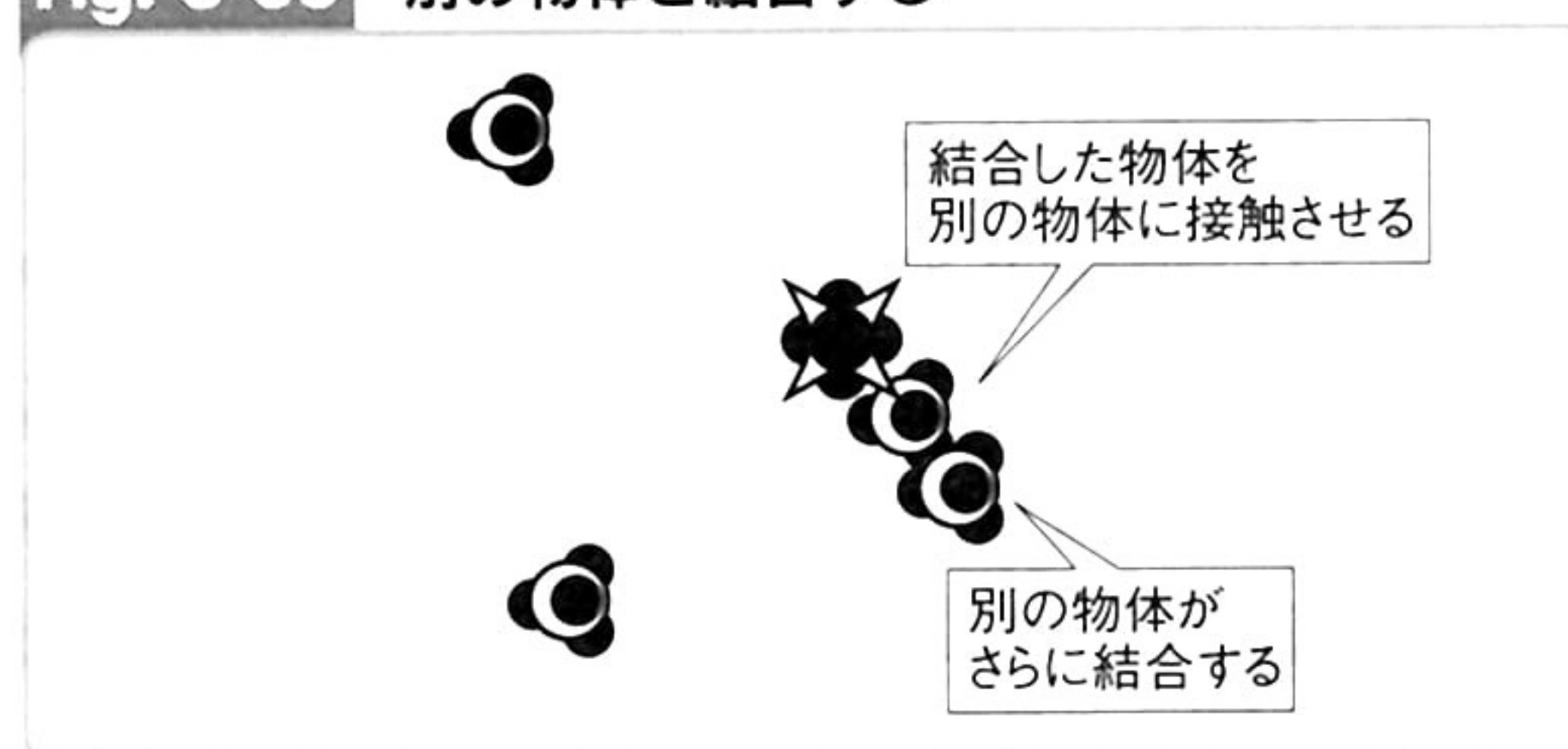



Fig. 3-39 多数の物体と結合する

Fig. 3-40 物体を分離する

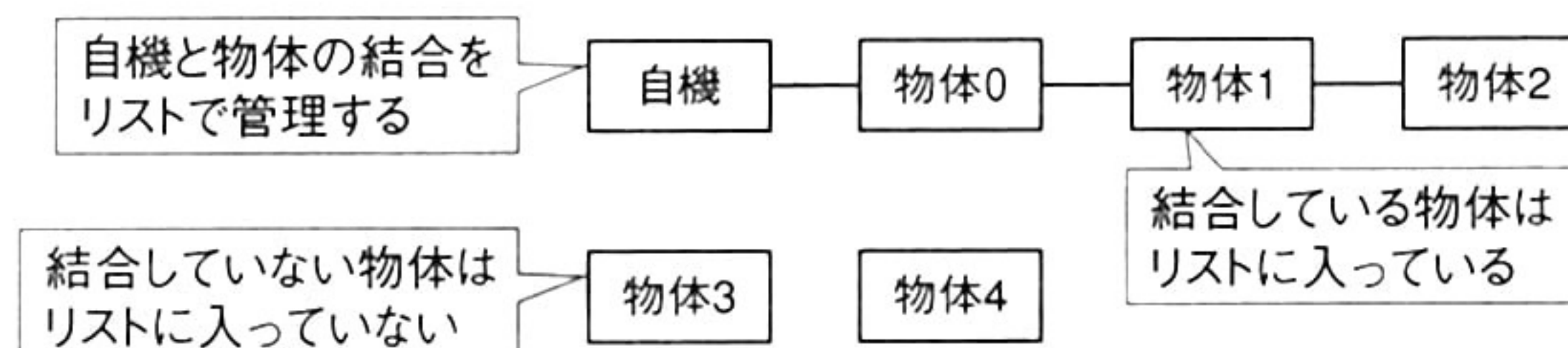
とミスになってしまうので、敵を避けつつ、適切な場所に物体を結合する必要があります。結合した物体の数が増えるにつれて、思った場所に物体が結合しなかったり、余分な物体が結合してしまったりすることが、このゲームの難しさであり、面白さでもあります。

アルゴリズム

結合して形を作るアクションを実現するポイントは、自機と物体の結合を管理する方法です。自機と結合した物体を一緒に動かしたり、結合したのとは逆の順序で物体を分離したりといった処理を行うには、結合の管理方法を工夫する必要があります。

本書では、自機と物体の結合をリストで管理することにしました (Fig. 3-41)。リストの先頭は自機で、その後に結合した物体が続きます。

結合をリストで管理することの利点は、結合できる物体の数に上限がないことです。画面上に出現させられるかぎり、いくつでも物体を結合することができます。上限を設けてもかまわなければ、スタックなどのデータ構造を使ってもよいでしょう。

Fig. 3-41 結合をリストで管理する

物体を結合する

新しく物体を結合するときには、自機および結合しているすべての物体について、まだ結合していない物体との間で当たり判定処理を行います (Fig. 3-42)。自機またはいずれかの物体が接触したときには、新しい物体を結合します。

新しい物体はリストの末尾に追加します (Fig. 3-43)。接触した物体の隣ではなく、リストの末尾に追加するのは、ボタンを押したときに最後に結合した物体を分離するためです。

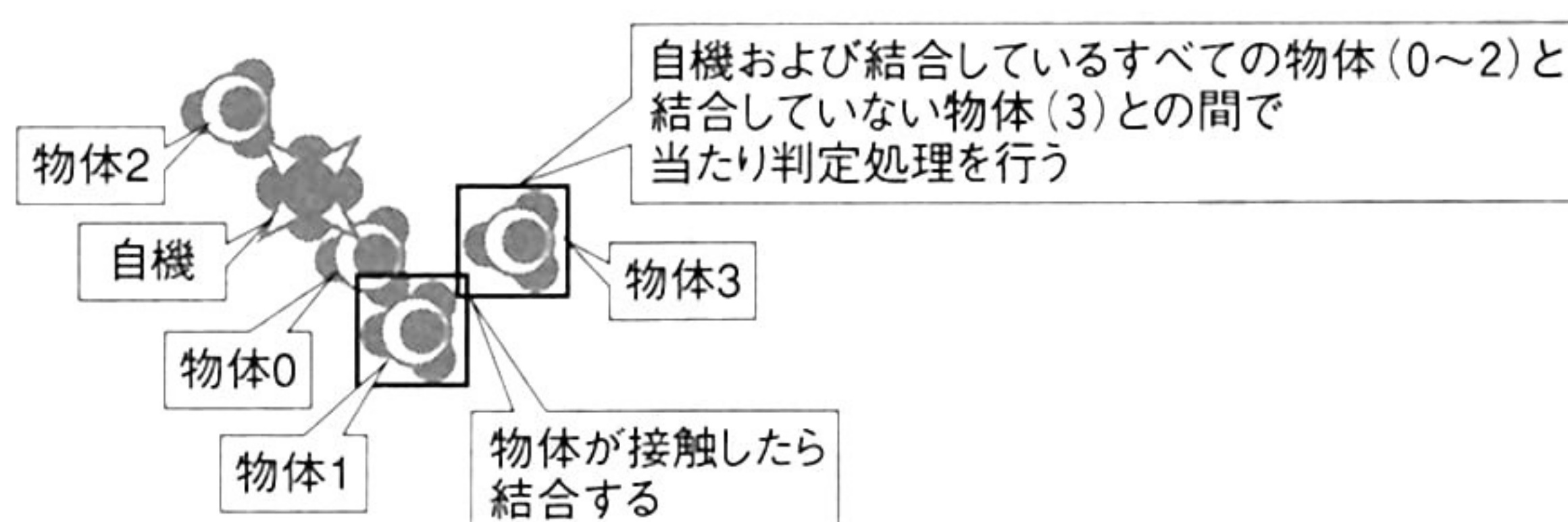
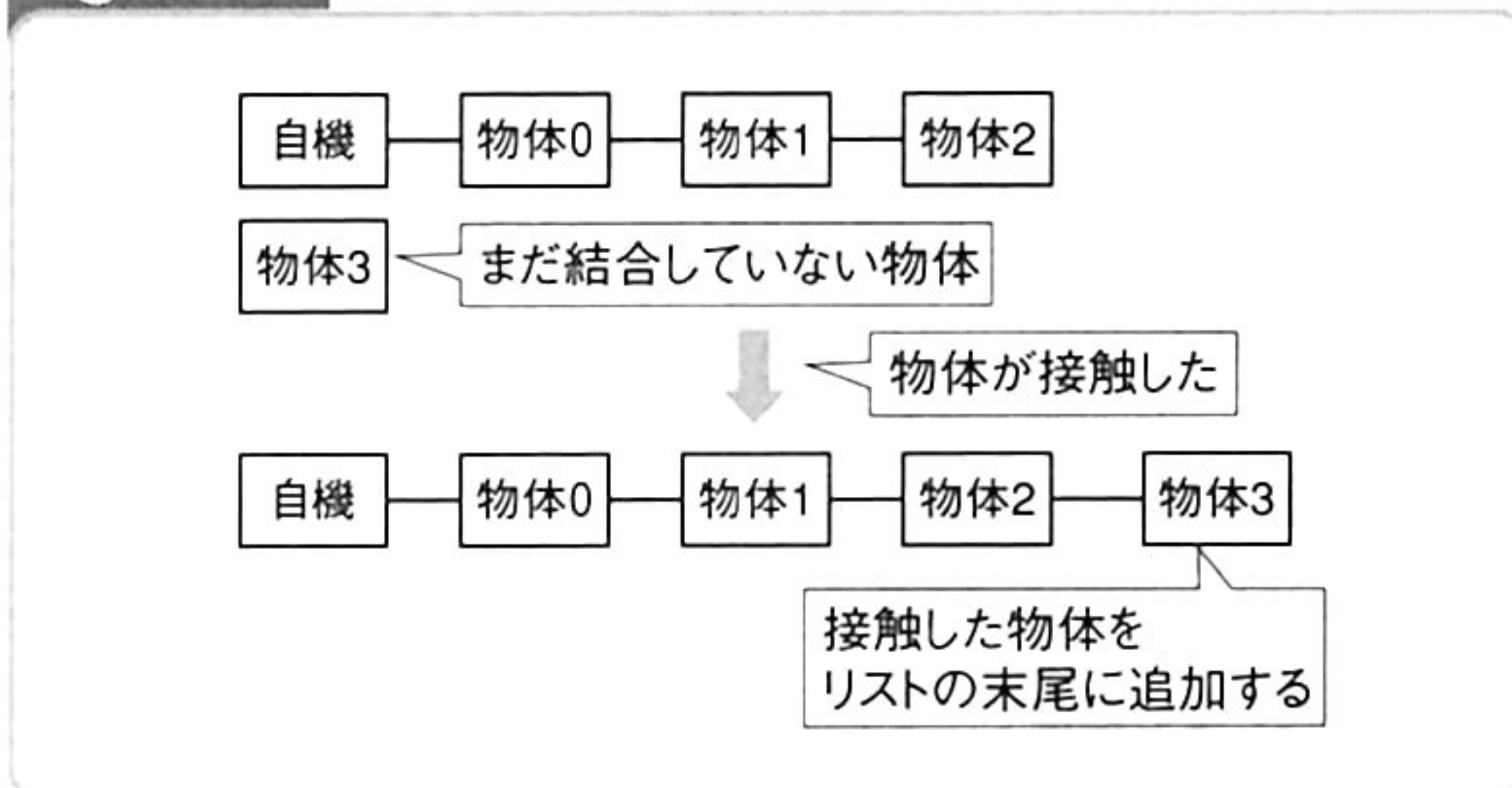
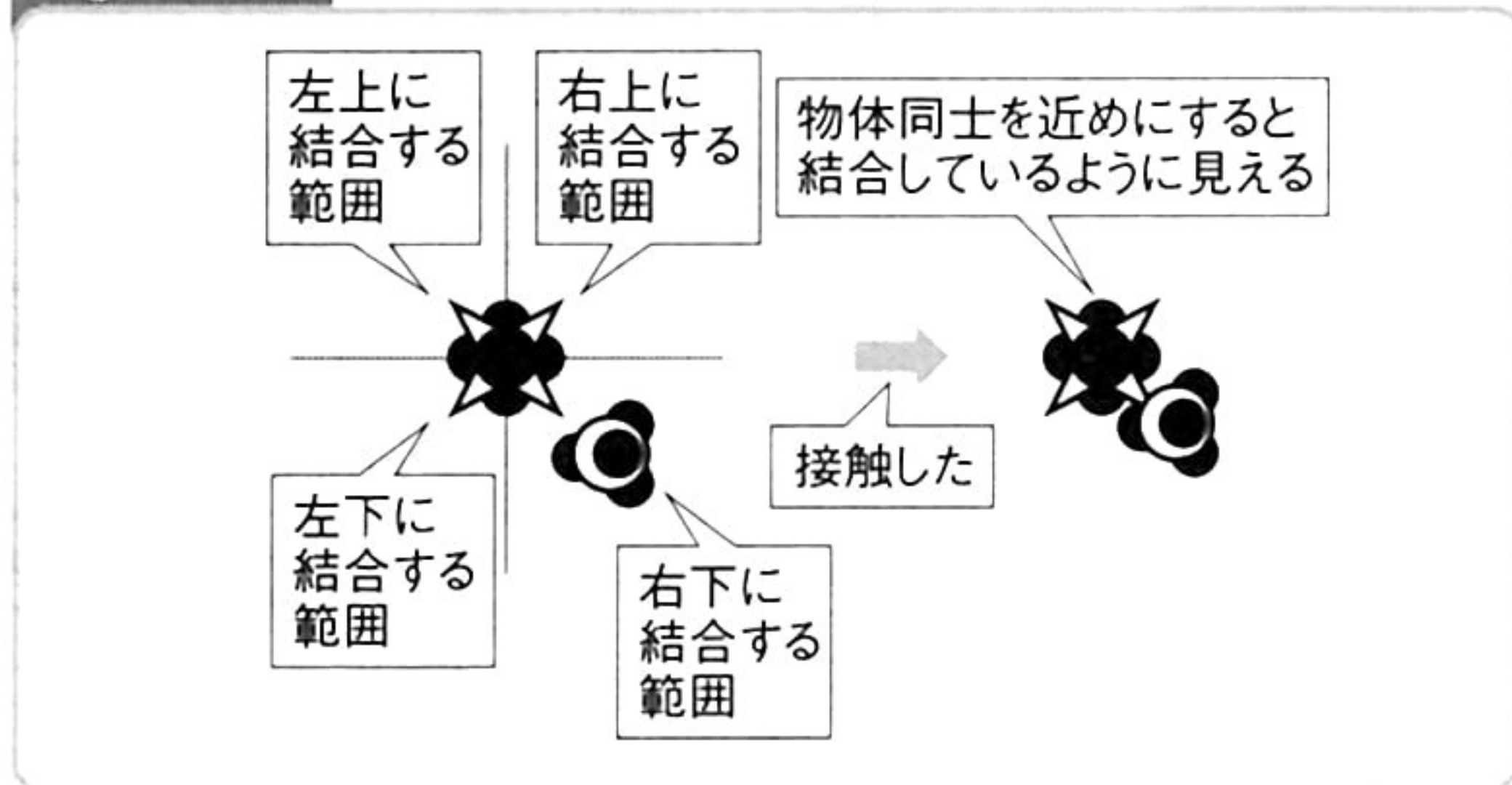
次に、新しい物体の座標を調整します (Fig. 3-44)。接触した物体に対して、どの方向から接

Fig. 3-42 物体の当たり判定処理

Fig. 3-43 新しい物体の追加

自機 — 物体0 — 物体1 — 物体2
物体3　まだ結合していない物体
物体が接触した
自機 — 物体0 — 物体1 — 物体2 — 物体3
接触した物体を
リストの末尾に追加する

Fig. 3-44 新しい物体の座標を調整する

左上に
結合する
範囲
右上に
結合する
範囲
物体同士を近めにすると
結合しているように見える
接触した
左下に
結合する
範囲
右下に
結合する
範囲

触したのかによって、左上・左下・右上・右下のいずれかに配置します。少し物体同士を近めに配置すると、結合しているように見えます。

物体の移動と分離

自機を動かしたときには、リストにあるすべての物体を、自機と同じ速度で動かします（Fig. 3-45）。自機と同じ速度を、物体の座標に加算すれば、結合した物体が自機と一緒に動いているように見えます。

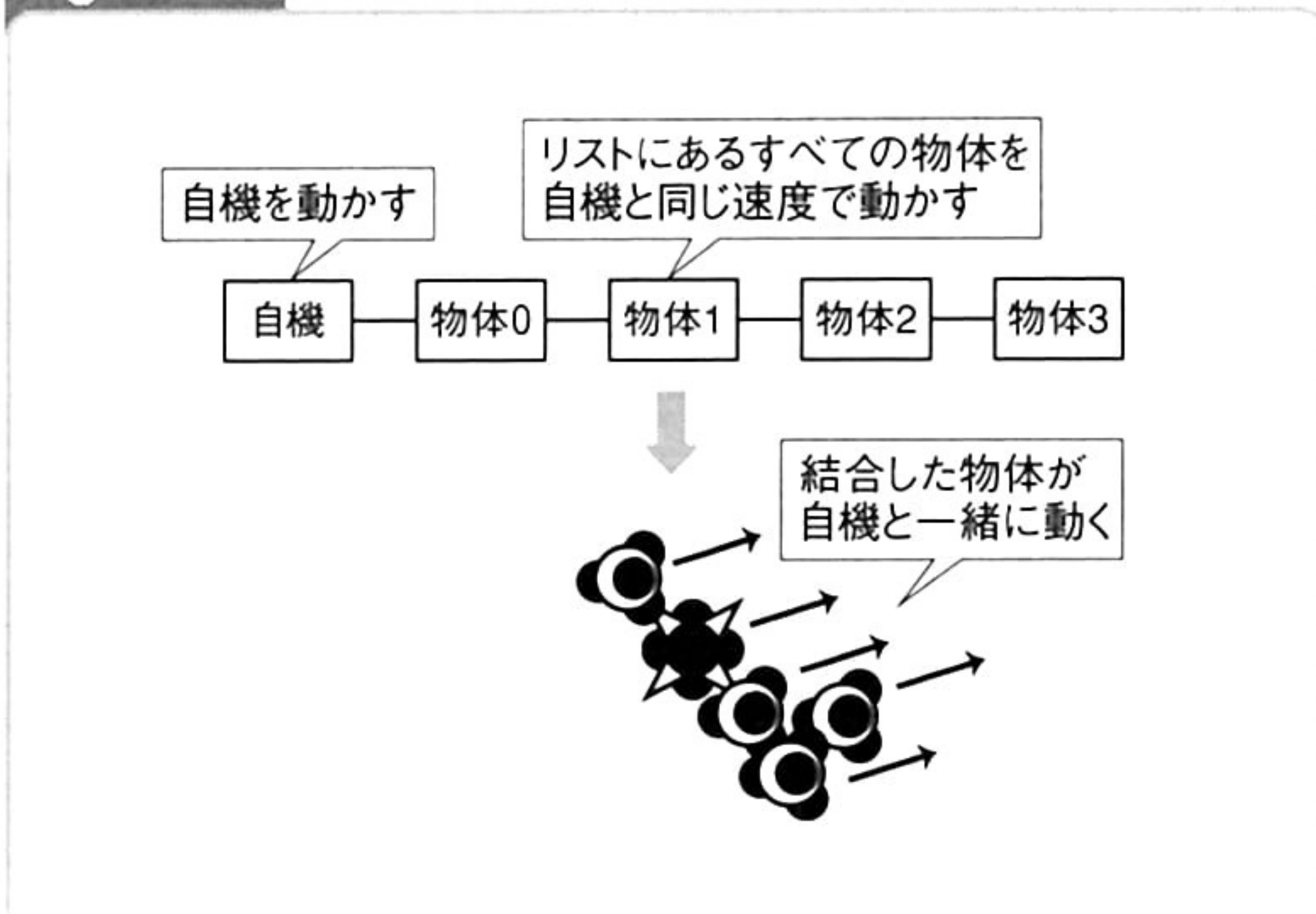
ボタンを入力したときには、最後に結合した物体を分離します（Fig. 3-46）。リストの末尾にある物体が、最後に結合した物体です。物体をリストから除去して、自機から離れるような速度を設定すれば、物体を分離することができます。

分離した物体は、分離した瞬間には自機や結合した物体の近くにいるため、再び接触しやすくなっています。分離したはずの物体が、すぐにまた結合してしまうと、操作がしにくくなります。これを回避する1つの方法は、一度分離した物体を、再び結合しないようにしておくことです。

本書のサンプルでは、一度分離した物体は、その物体自身に結合しているという扱いにしました（Fig. 3-47）。これで、他の物体に接触しても、二度と結合することはありません。自機についても、他の物体に結合することはないので、自機自身に結合しているという扱いにしました。

Fig. 3-45 結合した物体を自機と一緒に動かす

自機を動かす
リストにあるすべての物体を
自機と同じ速度で動かす
自機 — 物体0 — 物体1 — 物体2 — 物体3
結合した物体が
自機と一緒に動く

Fig. 3-46 物体を分離する

自機 — 物体0 — 物体1 — 物体2 — 物体3
ボタンを押す
自機 — 物体0 — 物体1 — 物体2　物体3
リストの末尾にある
物体を分離する
最後に結合した
物体を分離する
自機から離れるように
速度を設定する

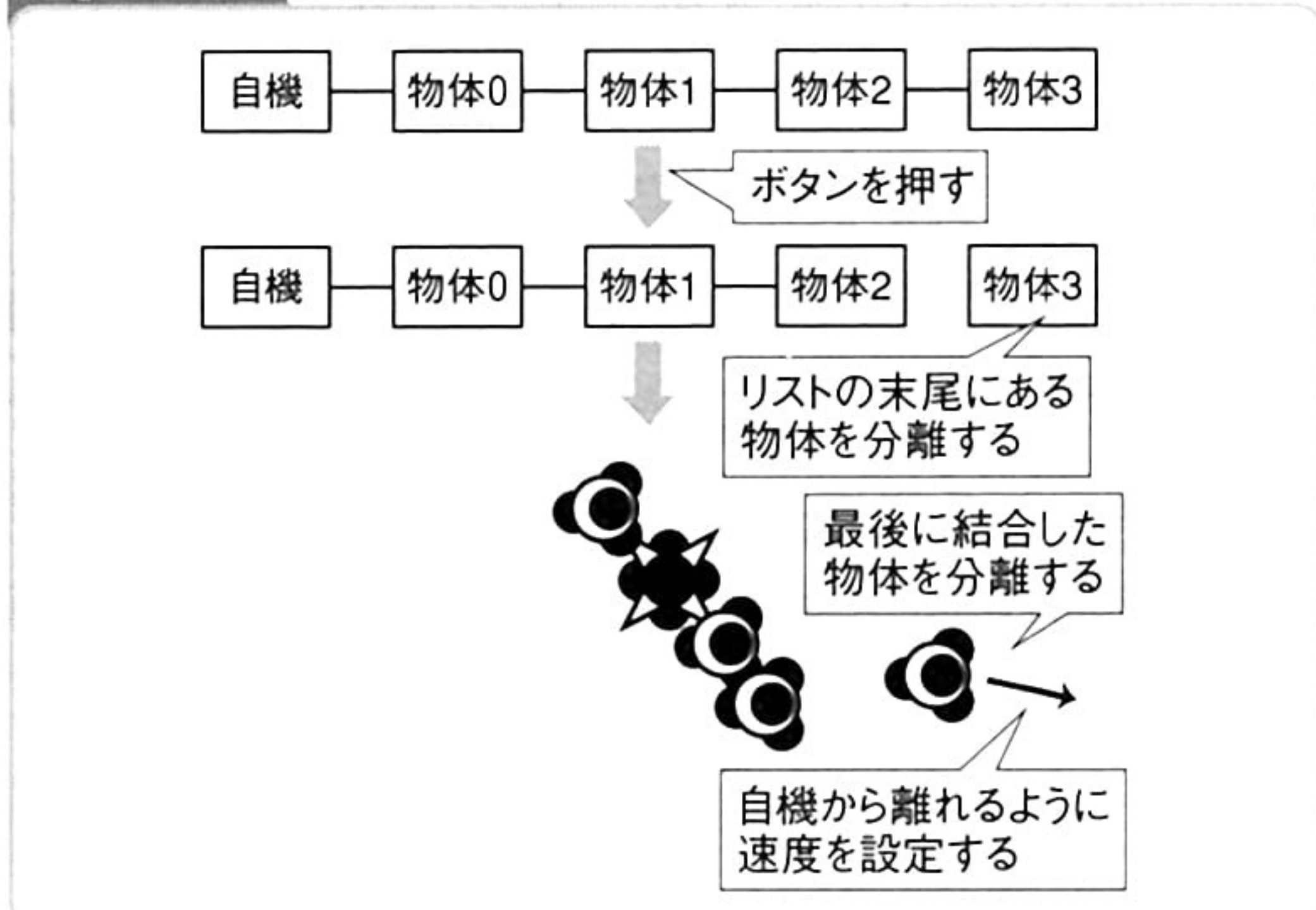
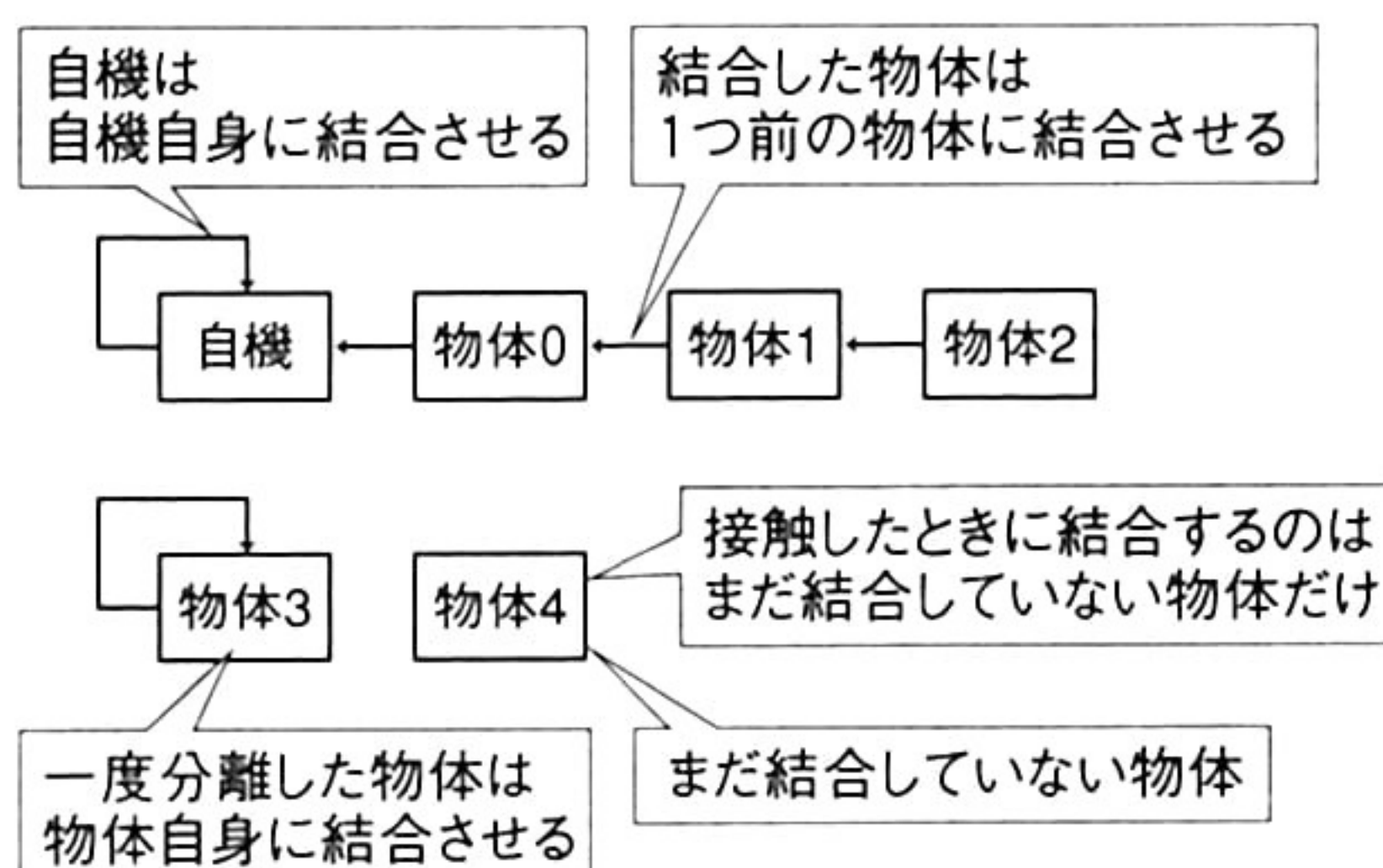



Fig. 3-47 他の物体に結合しない物体

プログラム

List 3-5は結合して形を作るプログラムです。物体のクラスと自機のクラスを掲載しました。

物体クラス（CLinkedShapeMoleculeクラス）は、画面を浮遊する物体のクラスです。リストを作るために、前後に結合している物体へのポインタ（Prev、Next）を用意しました。ここでは双方向連結リストを使いましたが、片方向連結リストを使うこともできます。

他の物体に結合していないときには、物体は一定の速度で直進し、画面外に出ると消滅します。結合しているときには、このクラスでは物体を動かさずに、自機クラスで動かします。

自機クラス（CLinkedShapeCursorクラス）は、プレイヤーが操作する自機のクラスです。このクラスは物体クラスから派生しています。

自機の移動処理では、レバー入力に応じて、自機を上下左右に動かします。また、リストを使って、自機に結合しているすべての物体を、自機と同じ速度で動かします。

さらに、すべての物体と、まだ結合していない物体との間で当たり判定処理を行います。接触した場合には、新しい物体を結合させるために、リストの末尾に追加したり、座標を調整したりします。

ボタンを入力したときには、リストの末尾にある物体を分離します。分離した物体をリストから除去するとともに、自機から離れるように速度を設定します。また、分離した物体が再び結合しないように、物体自身に結合しているような扱いにします。

List 3-5 結合して形を作る（CLinkedShapeMoleculeクラス、CLinkedShapeCursorクラス）

```
// 物体のクラス
class CLinkedShapeMolecule : public CMover {
public:

    // 前後に結合している物体
    CLinkedShapeMolecule* Prev;
    CLinkedShapeMolecule* Next;

    // 速度
```

```
    float VX, VY;

    //コンストラクタ
    CLinkedShapeMolecule(float x, float y)
    :    Prev(NULL), Next(NULL)
    {
        //グラフィックの設定
        Texture=Game->Texture[TEX_MOLECULE1];

        //座標をランダムに決める
        X=Rand.Int31()%2*(MAX_X+1)-1;
        Y=Rand.Int31()%2*(MAX_Y+1)-1;

        //自機に向かっていくように速度を決める
        VX=x-X;
        VY=y-Y;
        float v=sqrtf(VX*VX+VY*VY)/0.02f;
        if (v>0) {
            VX/=v;
            VY/=v;
        }
    }

    //移動処理
    virtual bool Move(const CInputState* is) {

        //自機や他の物体に結合していないときの処理
        if (!Prev || Prev==this) {

            //決められた速度で直進する
            X+=VX;
            Y+=VY;

            //グラフィックを回転させる
            Angle+=0.01f;

            //画面外に出たら消滅する
            return
                X>=-1 && X<=MAX_X &&
                Y>=-1 && Y<=MAX_Y;
        }

        //自機や他の物体に結合している間は消滅しない
        return true;
    }
};

//自機のクラス
//物体のクラスから派生する
```

```
class CLinkedShapeCursor :: public CLinkedShapeMolecule {
public:

    //前回のボタンの状態
    bool PrevButton;

    //コンストラクタ
    CLinkedShapeCursor()
    :    CLinkedShapeMolecule(0, 0), PrevButton(true)
    {
        //グラフィックの設定
        Texture=Game->Texture[TEX_MOLECULE0];

        //座標を画面の中心に合わせる
        X=(MAX_X-1)/2;
        Y=(MAX_Y-1)/2;

        //自機は他の物体に結合しないようにする
        //(自分自身に結合しているとする)
        Prev=this;
    }

    //移動処理
    virtual bool Move(const CInputState* is) {

        //移動スピード
        static const float speed=0.1f;

        //レバー入力に応じて速度を設定する
        VX=VY=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;
        if (is->Up) VY=-speed;
        if (is->Down) VY=speed;

        //画面外に出るときには速度を0にする
        if (X+VX<0 || X+VX>MAX_X-1) VX=0;
        if (Y+VY<0 || Y+VY>MAX_Y-1) VY=0;

        //最後に結合した物体を探す
        CLinkedShapeMolecule* tail=this;
        while (tail->Next) tail=tail->Next;

        //自機に結合しているすべての物体について処理する
        for (CLinkedShapeMolecule* m=this; m; m=m->Next) {

            //自機と一緒に動かす
            m->X+=VX;
            m->Y+=VY;
```

```
    //まだ結合していない物体との接触を調べる
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {

        //画面上にある物体のなかから1つを取得する
        CLinkedShapeMolecule*
            n=(CLinkedShapeMolecule*)i.Next();

        //当たり判定の大きさ、結合後の距離
        static const float hit=1.0f, link=0.6f;

        //ある物体が結合しておらず、
        //自機または結合した物体からの距離が一定値未満ならば、
        //その物体を結合させる
        if (
            !n->Prev &&
            abs(n->X-m->X)<hit && abs(n->Y-m->Y)<hit
        ) {
            //接触した方向に応じて、
            //左上・左下・右上・右下のいずれかに結合させる
            n->X=m->X+(n->X<m->X?-link:link);
            n->Y=m->Y+(n->Y<m->Y?-link:link);

            //回転を初期化する
            n->Angle=0;

            //最後に結合した物体として登録する
            n->Prev=tail;
            tail->Next=n;
            tail=n;
        }
    }
}

//ボタンを入力したときの処理
//ボタンを押しっぱなしにしたとき、
//次々に物体が分離してしまわないように、
//以前のボタンの入力状態を調べる
if (!PrevButton && is->Button[0]) {

    //自機に1個以上の物体が結合しているときの処理
    if (tail->Prev) {

        //分離した物体の速度を、
        //自機から遠ざかるように決める
        float
            vx=tail->X-X, vy=tail->Y-Y,
            v=sqrtf(vx*vx+vy*vy)/0.12f;
        if (v>0) {
```

```
                tail->VX=vx/v;
                tail->VY=vy/v;
            }

            //物体をリストから分離する
            tail->Prev->Next=NULL;

            //一度分離した物体は再び結合しないようにする
            //(物体自身に結合しているとする)
            tail->Prev=tail;
        }
    }

    //ボタンの入力状態を保存する
    PrevButton=is->Button[0];

    return true;
}
};
```

SAMPLE

「LINKED SHAPE」は「結合して形を作る」のサンプルです。レバーの上下左右(カーソルキーの上下左右)で自機が移動します。自機と物体を接触させると、物体と結合することができます。自機に結合している物体と、他の物体を接触させたときにも、結合が可能です。結合した物体は、自機と一緒に動きます。

ボタン0(Zキー)を押すと、最後に結合した物体が分離します。一度分離した物体に、再び結合することはできません。ボタンを押すたびに、結合したときとは逆の順番で物体が分離し、最後には自機だけの状態に戻ります。

LINKED SHAPE → p. 387

線で囲む

カーソルを動かして線を引くアクションです。線で囲んだ領域は塗りつぶすことができます。一定以上の領域で塗りつぶしたり、敵を囲んで倒したりすることが、ゲームの目的です。

最初はステージには何もなく、周囲だけに線が引かれている状態です(Fig. 3-48)。カーソルは線の上にあります。また、敵はステージのなかを自由に動き回っています。

カーソルはレバーで上下左右に動きます。ただし、線から外れることはできません。線に沿ってのみ動きます(Fig. 3-49)。

ボタンを押しながらレバーを操作すると、線から外れることができます(Fig. 3-50)。カーソ

ルが通った後には、新しい線が引かれていきます。図では、元からあった古い線を黒色で、新しく引いた線を灰色で示しました。

カーソルが古い線に到達すると、線で囲んだ領域が塗りつぶされます（Fig. 3-51）。新しい線を引くことによって、2つの領域ができますが、塗りつぶされるのは敵がいない方の領域です。

領域を塗りつぶすと、今度は塗りつぶした領域の周囲が古い線（黒色）になり、その上をカーソルが動けるようになります（Fig. 3-52）。これを繰り返して、ステージを次々に塗りつぶし、敵を追い詰めていきます（Fig. 3-53）。

線で囲むゲームには『クイックス』や『ヴォルフィード』などがあります。このゲームでは、カーソルを動かして線を引き、領域を囲んで塗りつぶすことが目的です。ステージ内の一定割合以上の領域を塗りつぶすとクリアとなります。

このゲームには、プレイヤーを妨害するさまざまな敵が出現します。塗りつぶされていない領域を自由に動き回る敵や、引いた線に沿って追いかけてくる敵がいます。敵にカーソルが接触したり、引いている途中の線が接触したりすると、ミスになってしまいます。敵に接触せずに、いかに効率よく領域を塗りつぶしていくかが、このゲームの面白さです。

『クイックス』に似たゲームには『ギャルズパニック』シリーズがあります。このゲームでは、領域を塗りつぶすと、背後にあるグラフィック（大人向け）が見えるようになっています。

『ダンシングアイ』も同様のゲームですが、こちらはステージが3次元の筒状になっています。

Fig. 3-48 最初の状態

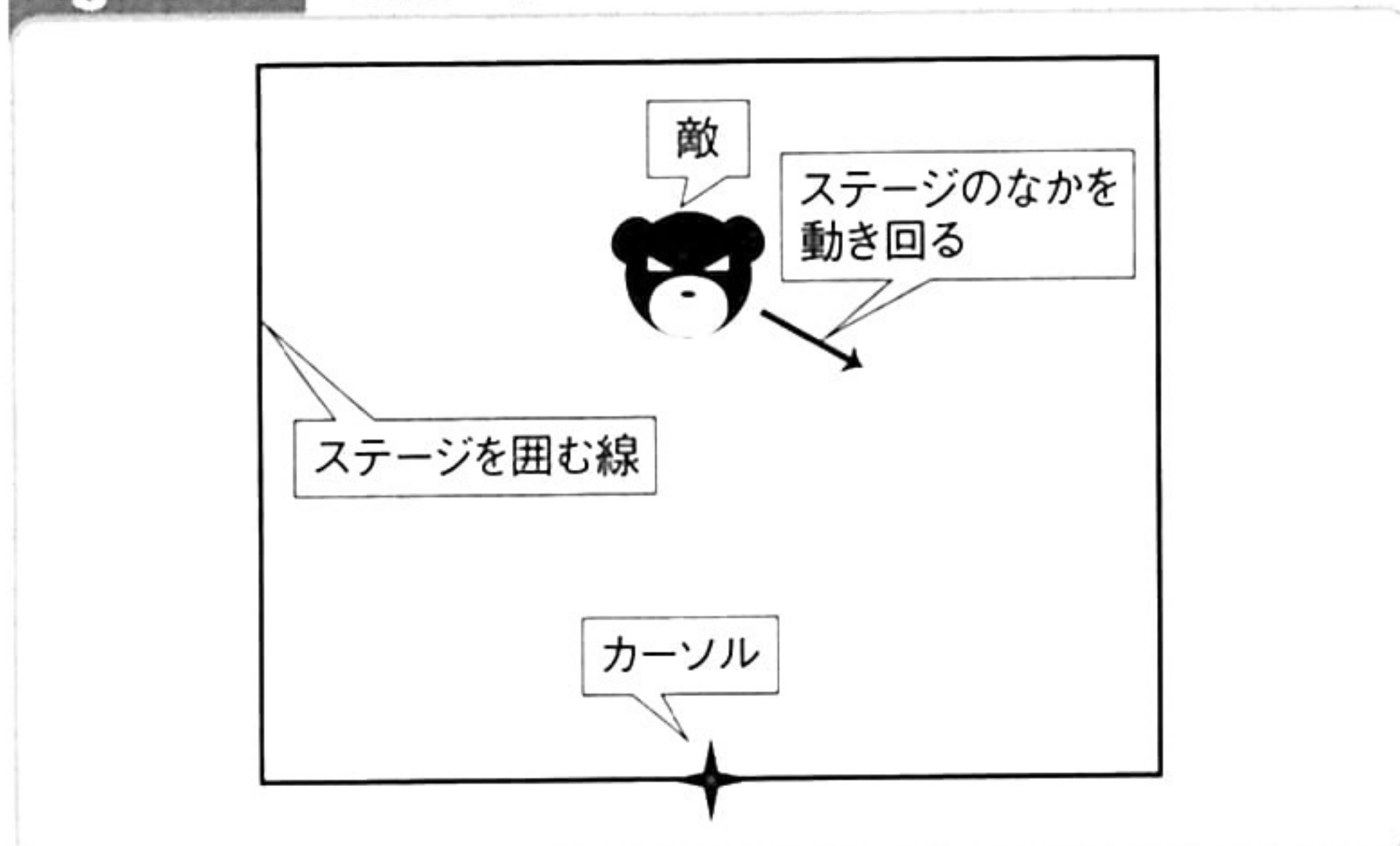



Fig. 3-49 線に沿って動くカーソル

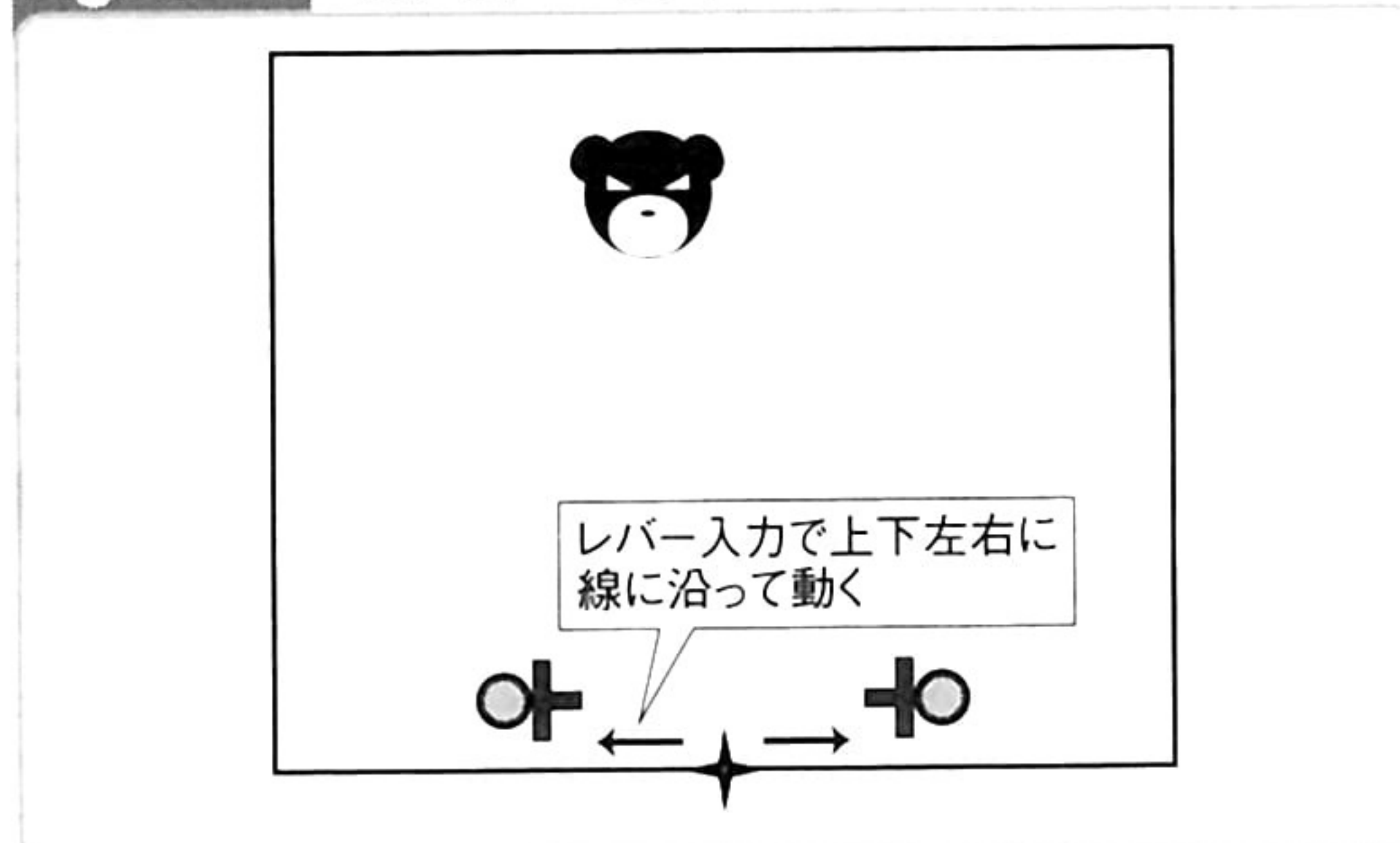



Fig. 3-50 線を引きながら動くカーソル

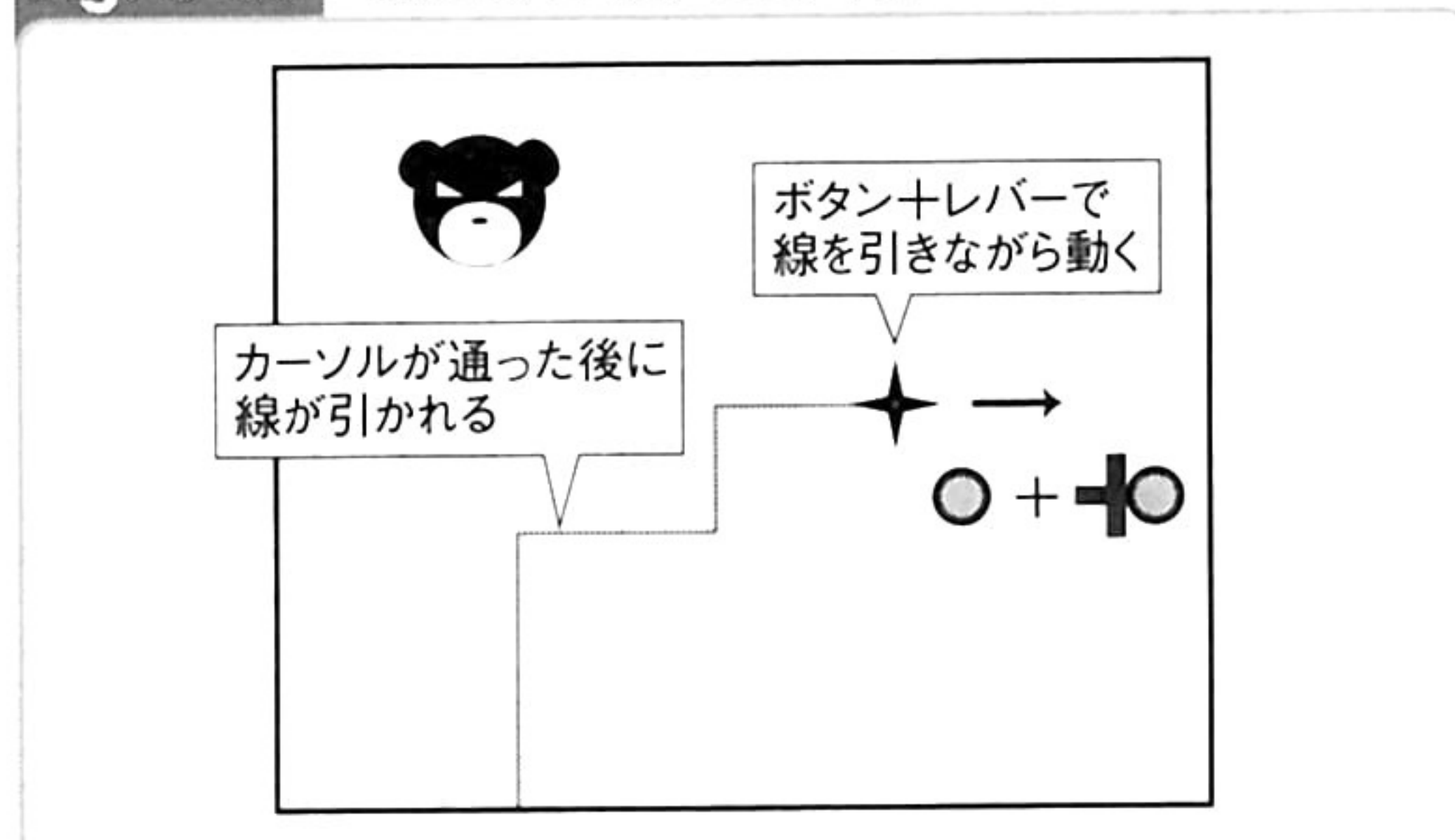



Fig. 3-51 線で囲んだ領域を塗りつぶす

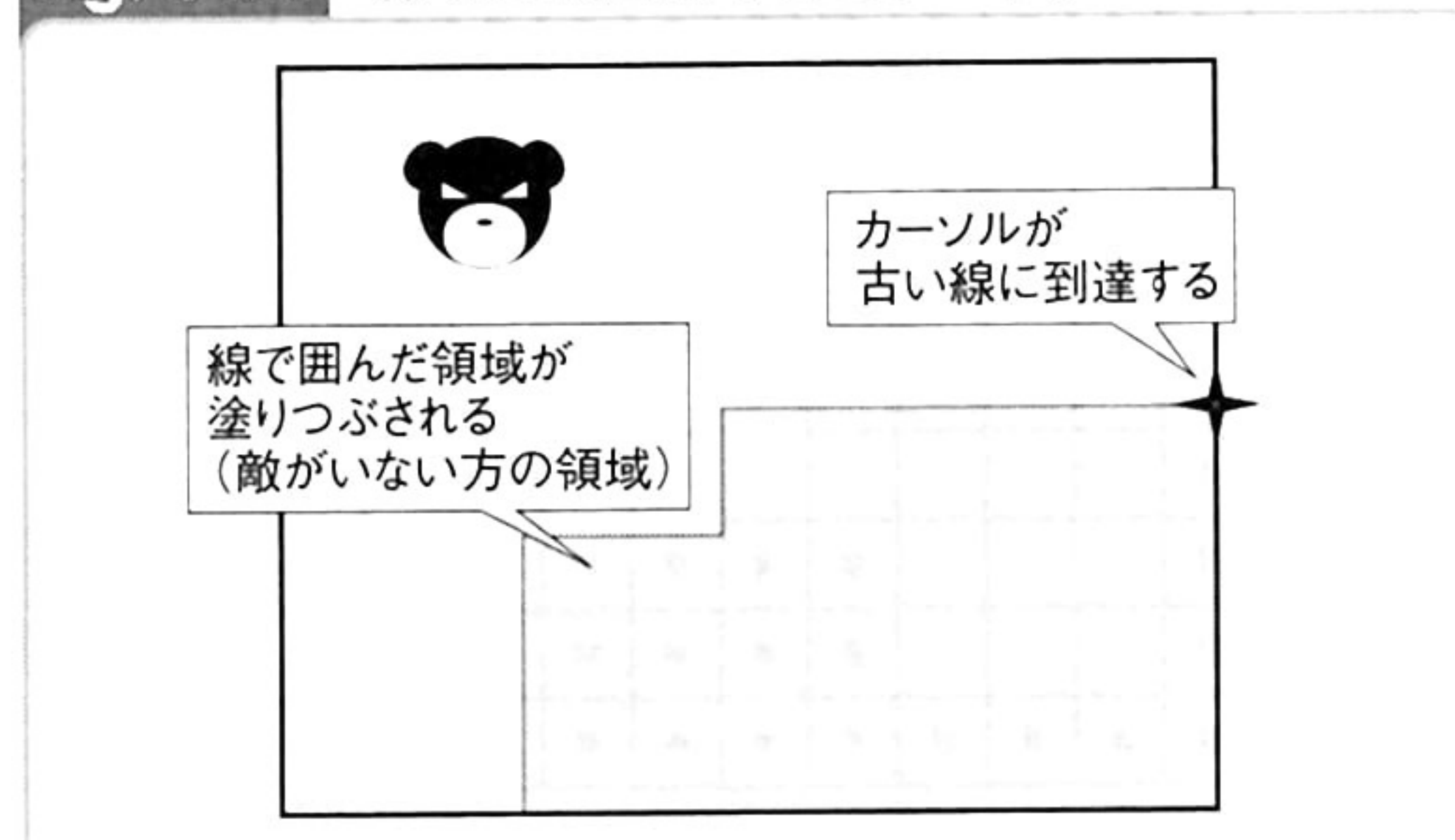



Fig. 3-52 塗りつぶした領域の周囲を動くカーソル

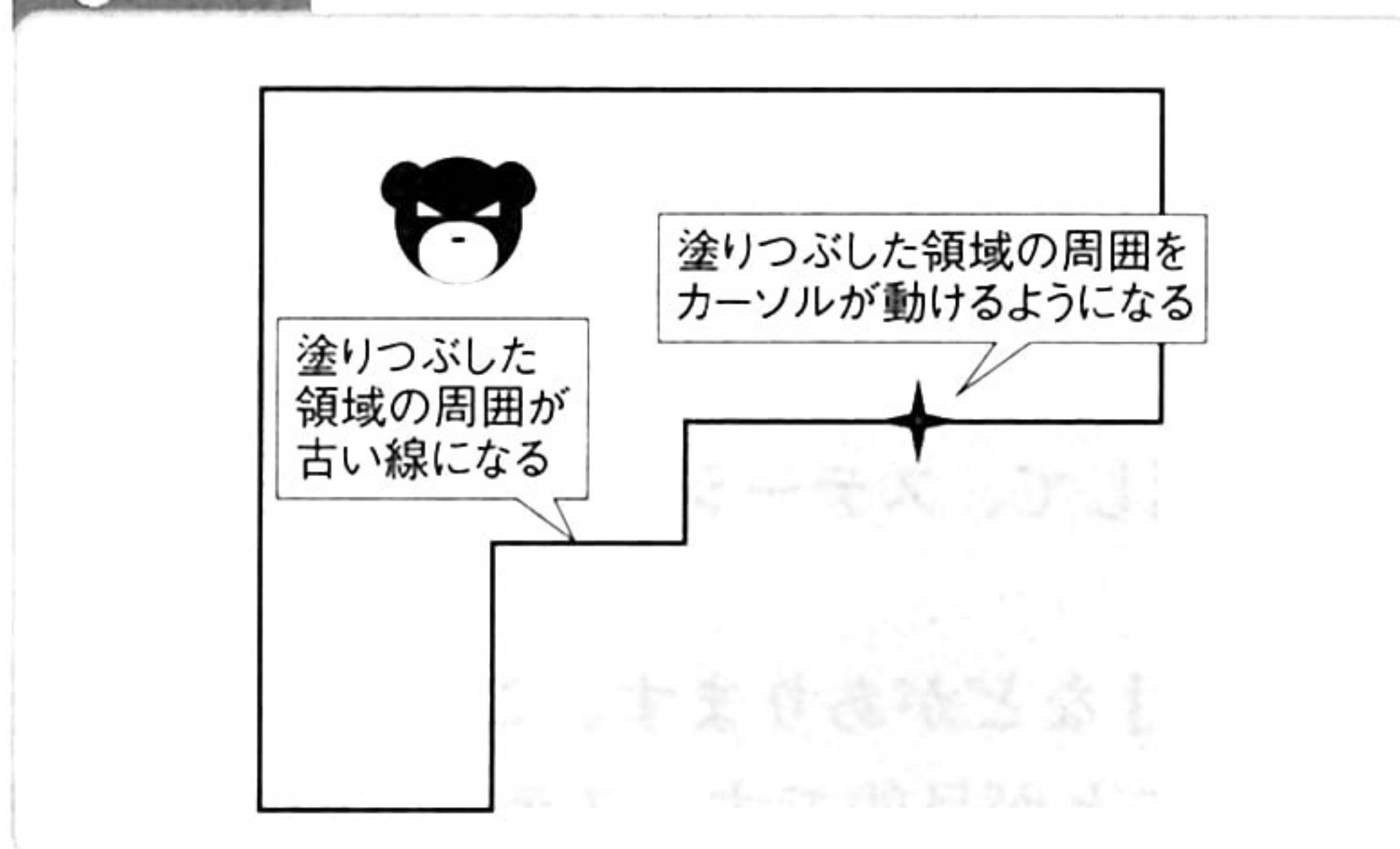
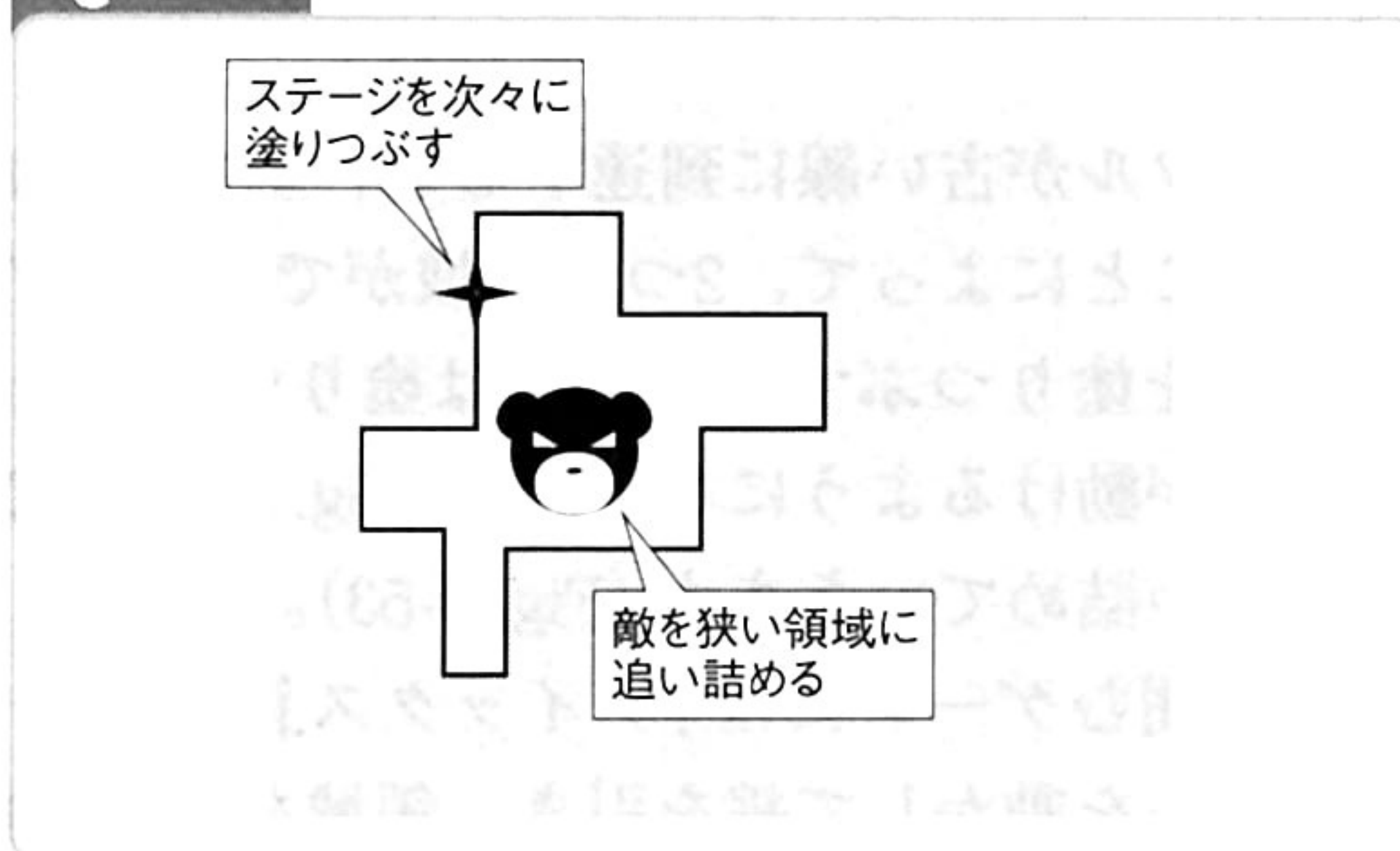



Fig. 3-53 ステージを次々に塗りつぶす

アルゴリズム

線で囲むアクションを実現するには、ステージをセルで表現します（Fig. 3-54）。古い線（黒色）は「#」、新しい線（灰色）は「+」、塗りつぶした領域は「*」で表します。

カーソルを線に沿って動かすには、カーソルの移動方向にあるセルを調べます（Fig. 3-55）。ここでは1つ先のセルと、2つ先のセルを調べます。両方のセルが古い線（#）ならば、カーソルを2つ先に進めます。

カーソルを1つ先のセルに進めるのではなく、2つ先のセルに進めるのは、線同士を隣接させないためです（Fig. 3-56）。カーソルをセル1個単位で動かすと、平行する線を引いたときに、線同士が隣接（接触）してしまうことがあります。

この場合、カーソルがどちらの線に沿って動けばよいのかが判断しにくくなります。また、囲まれた領域が3個以上できてしまうことがあるため、塗りつぶしの処理が複雑になります。カーソルをセル2個単位で動かすことにしておくと、線同士が隣接しなくなるため、こういった問題を回避することができます。

Fig. 3-54 ステージをセルで表現する

#	#	#	#	#	#	#	#	#	#	#
#				+						#
#			+	+						#
#			+							#
#		+	+			#	#	#	#	#
#						#	*	*	*	*
#				#	#	#	*	*	*	*
#				#	*	*	*	*	*	*
#	#	#	#	#	*	*	*	*	*	*

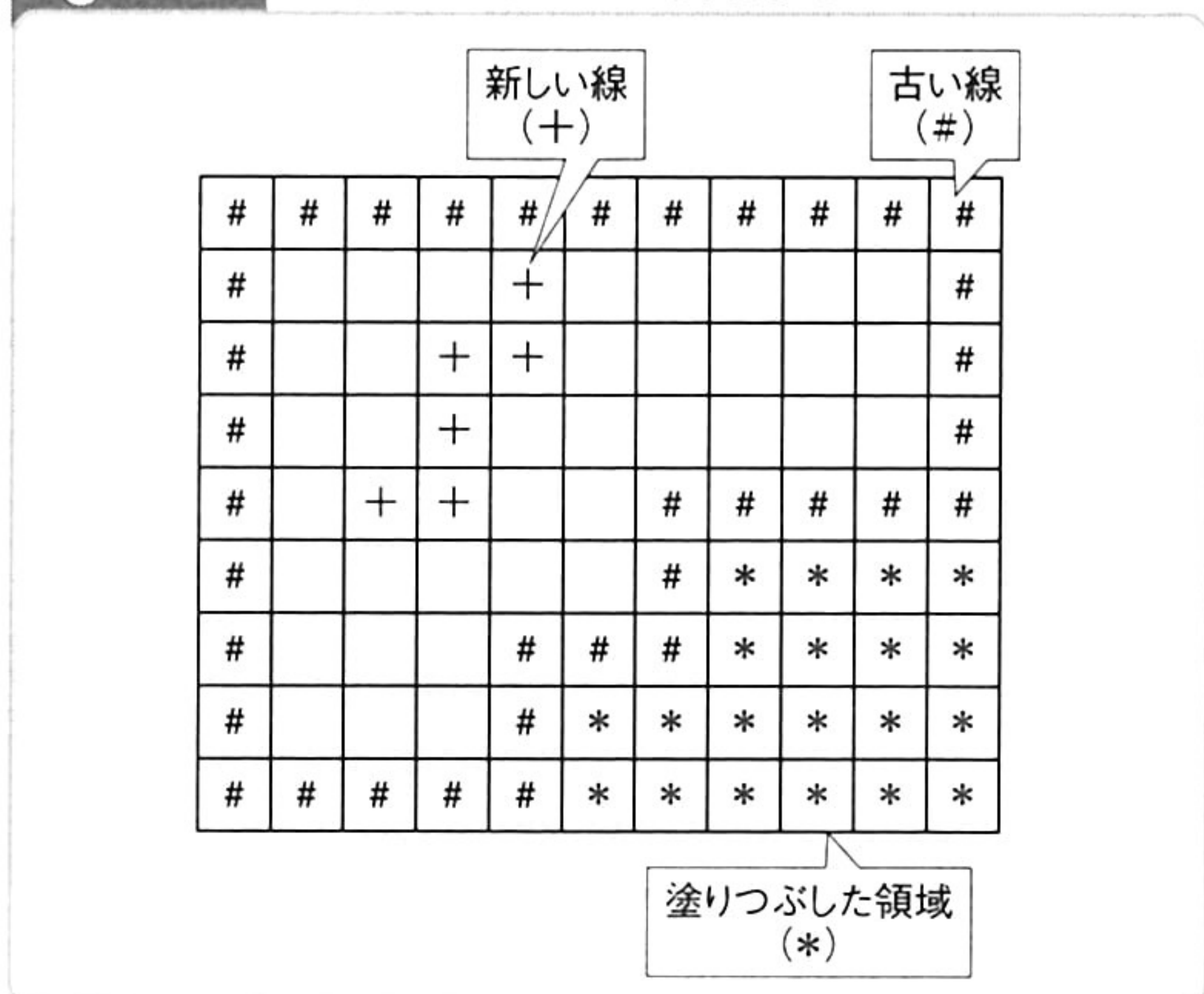



Fig. 3-55 カーソルを線に沿って動かす

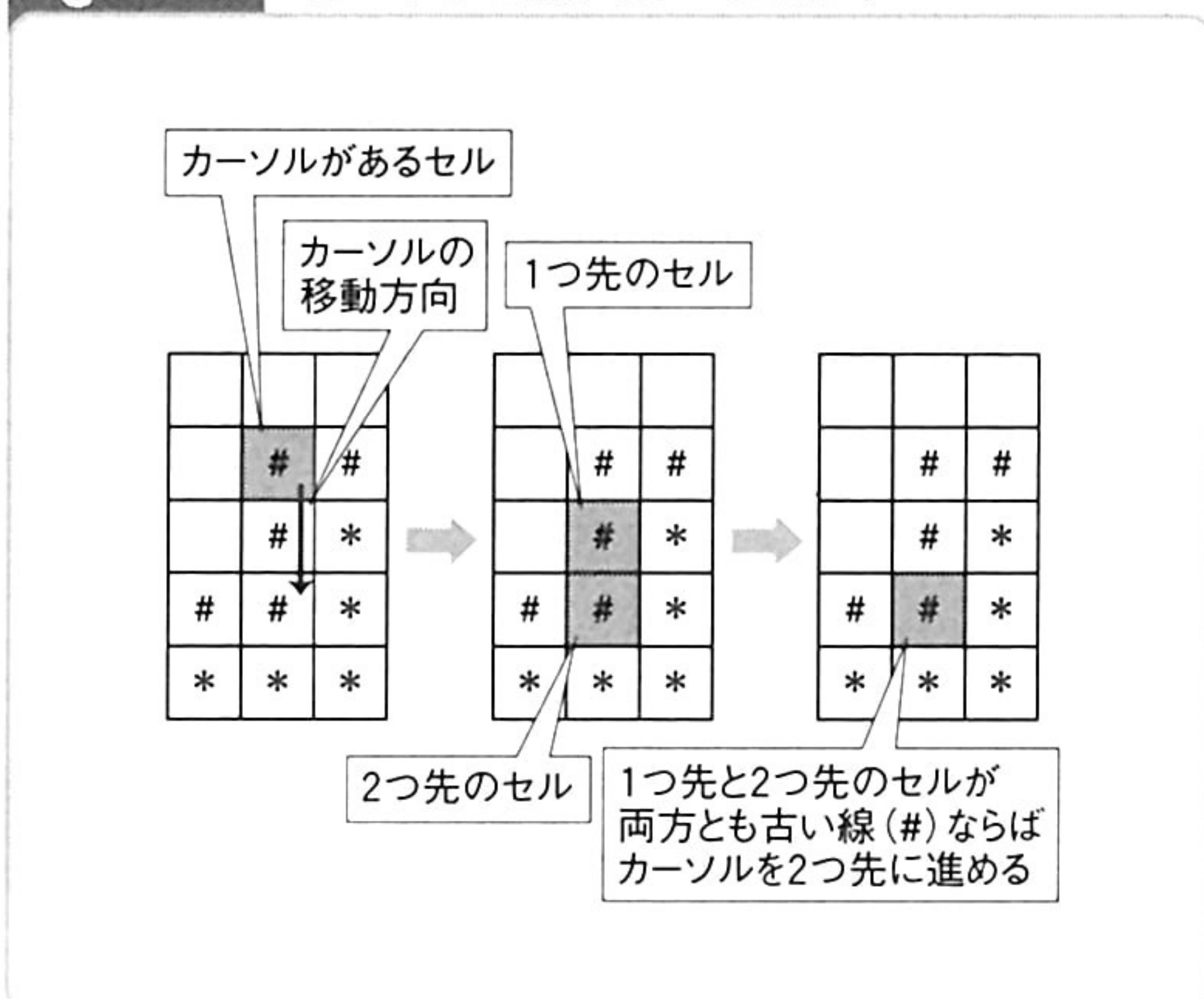
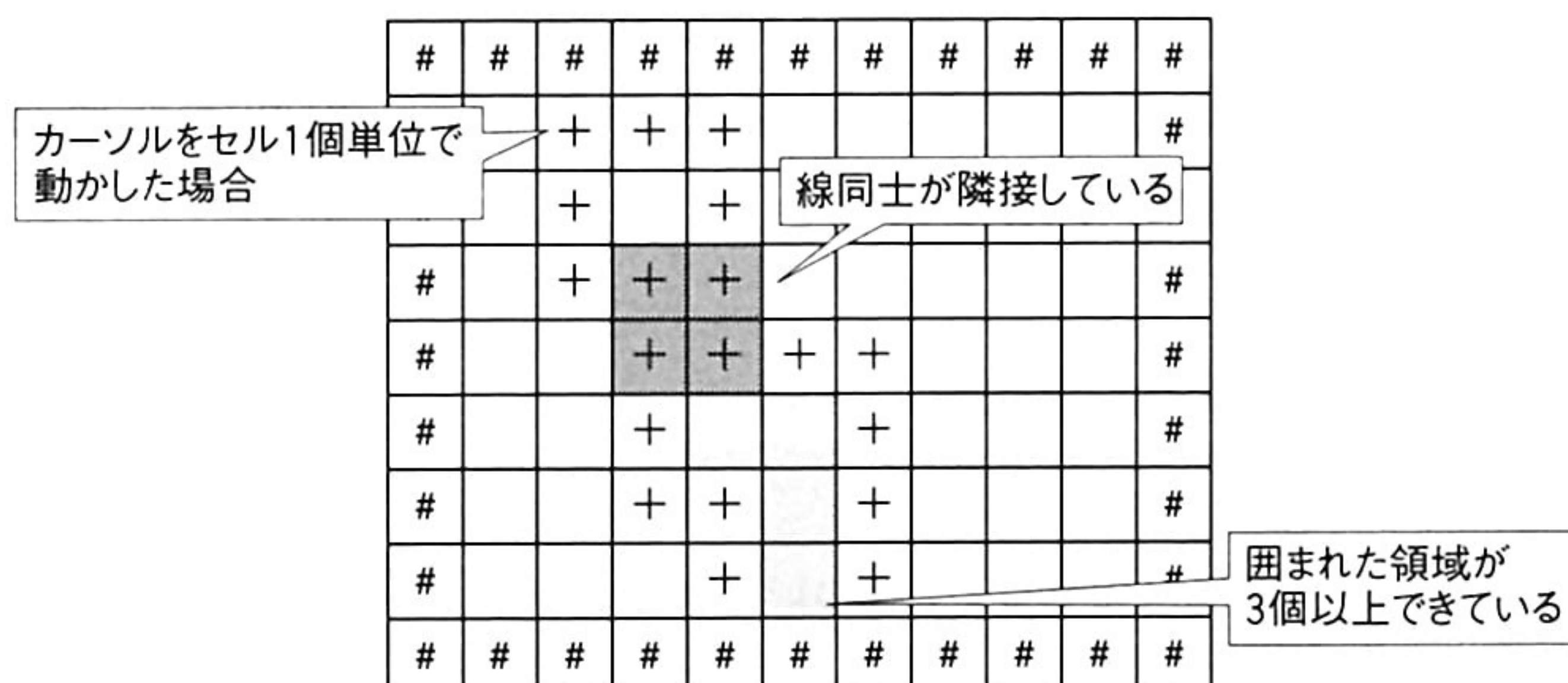



Fig. 3-56 カーソルを2つ先のセルに進める理由

新しい線を引く

ボタンを押しているときには、カーソルを動かして新しい線を引くことができます。カーソルの1つ先のセルと、2つ先のセルを調べて、両方のセルが空ならば、新しい線を引きます(Fig. 3-57)。1つ先のセルと2つ先のセルを、新しい線を表す「+」にします。

ボタンを押しているときに、1つ先のセルが空で、2つ先のセルが古い線(#)ならば、領域を閉じます(Fig. 3-58)。囲んだ領域を塗りつぶす方法については、「囲んだ領域を塗りつぶす」(→p. 179)で解説します。

Fig. 3-57 新しい線を引く

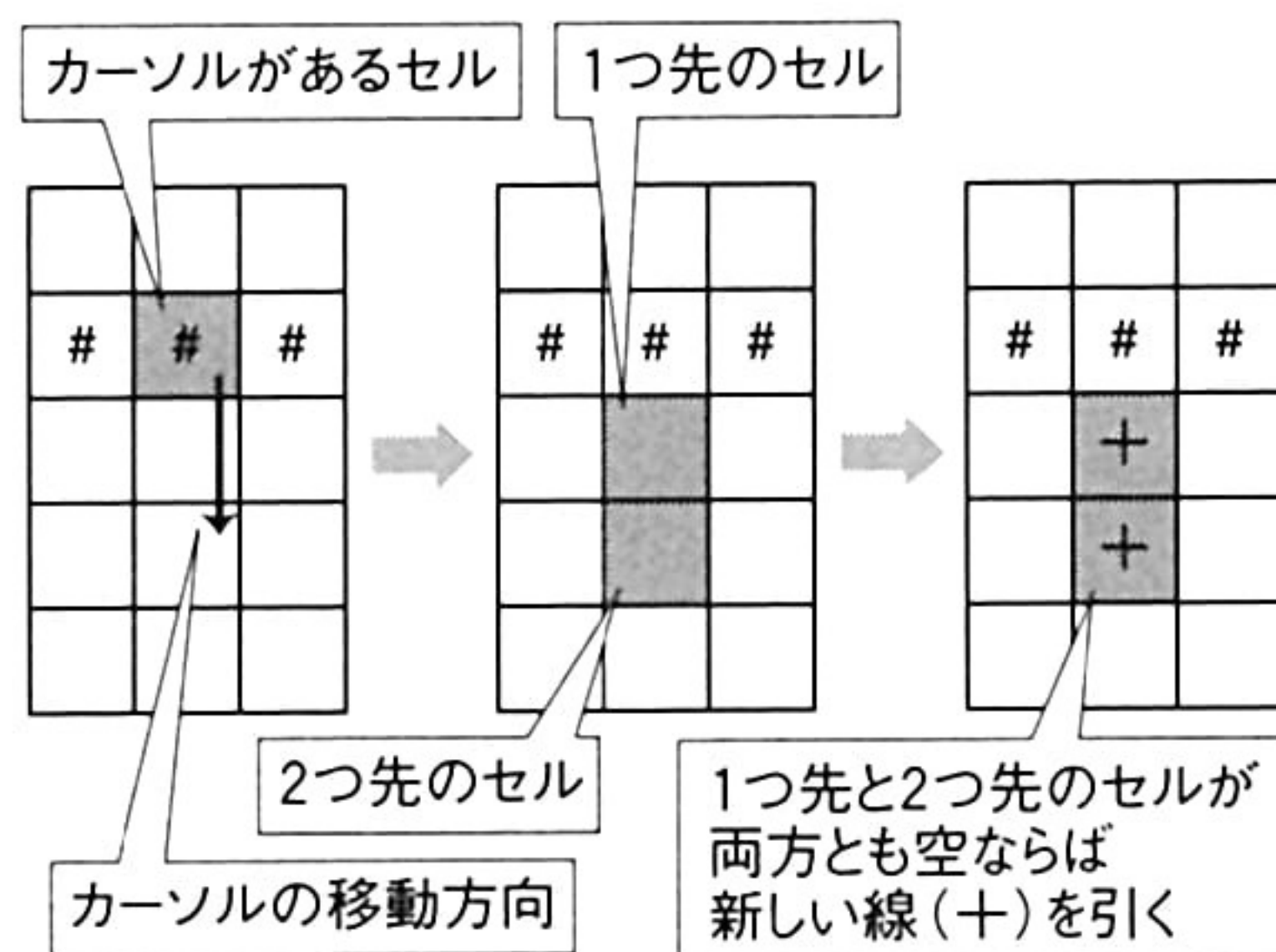
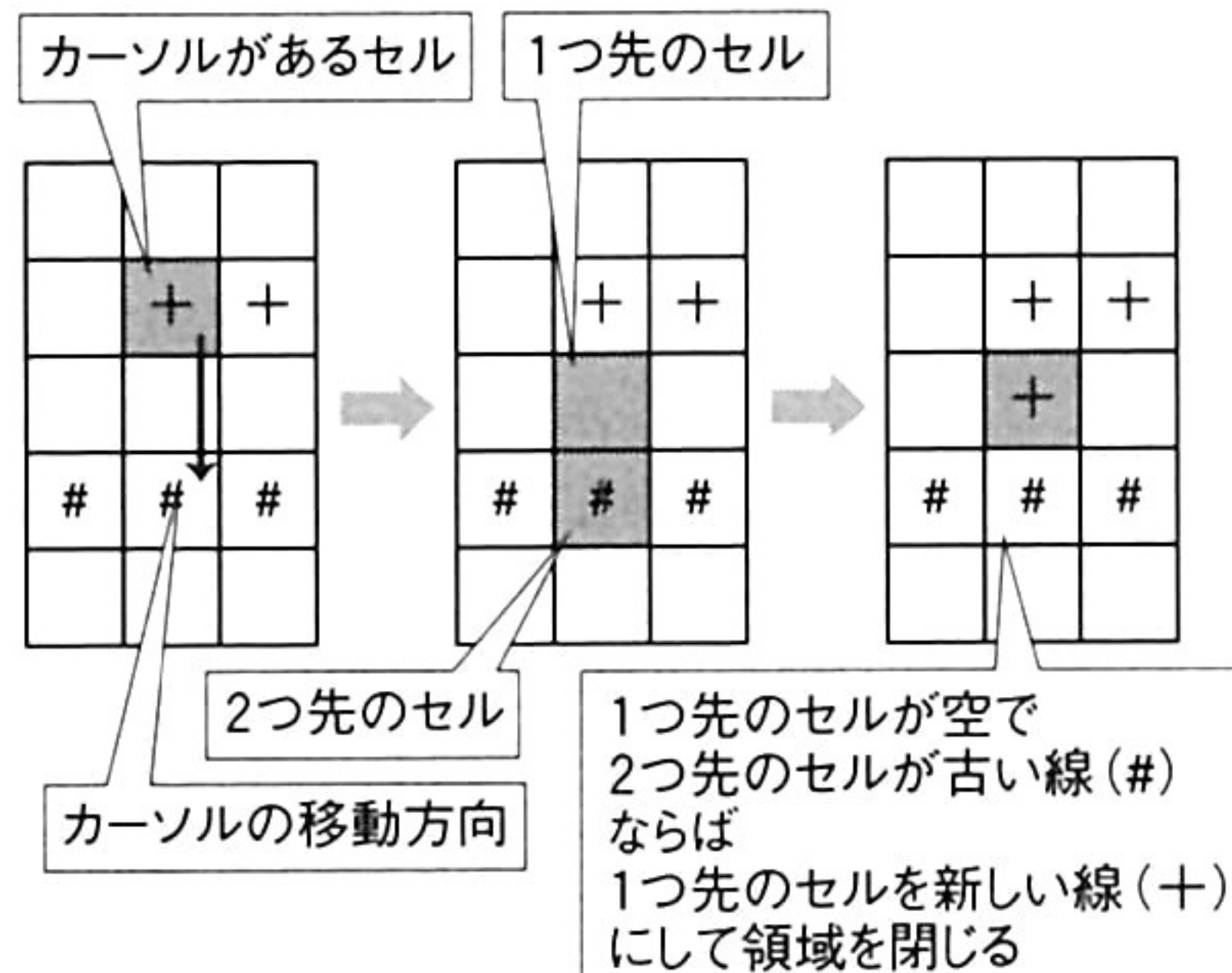



Fig. 3-58 領域を閉じる

プログラム

List 3-6は線で囲むプログラムです。カーソルの移動処理を掲載しました。

移動処理(Move関数)では、レバー入力に応じて、カーソルの移動方向を決めます。次に、カーソルの1つ先のセルと、2つ先のセルを調べます。

ボタンを押しているときに、1つ先と2つ先のセルが空ならば、新しい線を引きます。1つ先

が空で、2つ先が古い線ならば、領域を閉じます。ボタンを押していないときには、1つ先と2つ先のセルが両方とも古い線の場合だけ、カーソルを動かします。

カーソルはセル2個分ずつ動かします。これは前述のように、線同士が隣接することや、囲まれた領域が3個以上できることを防ぐためです。

List 3-6　線で囲む（CEnclosedAreaCursorクラス）

```
// カーソルの移動処理
bool CEnclosedAreaCursor::Move(const CInputState* is) {

    // レバーの入力に応じて、移動方向を設定する
    int vx=0, vy=0;
    if (is->Left) vx=-1; else
    if (is->Right) vx=1; else
    if (is->Up) vy=-1; else
    if (is->Down) vy=1;

    // 1つ先のセルと、2つ先のセルを取得する
    char
        c1=Cell->Get(CX+vx, CY+vy),
        c2=Cell->Get(CX+vx*2, CY+vy*2);

    // ボタンを押しているときの処理
    if (is->Button[0]) {

        // 1つ先のセルが空で、
        // 2つ先のセルも空のときには、線を引く
        if (c1==' ' && c2==' ') {

            // 1つ先のセルと2つ先のセルを線にする
            Cell->Set(CX+vx, CY+vy, '+');
            Cell->Set(CX+vx*2, CY+vy*2, '+');
        } else

        // 1つ先のセルが空で、
        // 2つ先のセルが古い線のときには、 領域を閉じる
        if (c1==' ' && c2=='#') {

            // 1つ先のセルを線にする
            Cell->Set(CX+vx, CY+vy, '+');

            // ...（中略）...

        } else

        // それ以外の場合には、カーソルを移動させない
        {
            vx=vy=0;
        }
```

```
    } else

    //ボタンを押していないときの処理
    {
        //1つ先のセルと2つ先のセルが、
        //ともに古い線であるとき以外には、
        //カーソルを移動させない
        if (c1!='#' || c2!='#') {
            vx=vy=0;
        }
    }

    //カーソルのセル座標を更新する
    //線同士が隣接しないようにするために、
    //セルを2つずつ移動する
    CX+=vx*2;
    CY+=vy*2;

    return true;
}
```

SAMPLE

「ENCLOSED AREA」は「線で囲む」「囲んだ領域を塗りつぶす」「囲まれた領域を避けて動く敵」のサンプルです。

レバーの上下左右(カーソルキーの上下左右)でカーソルが移動します。ボタンを押していないときには、カーソルは古い線(黒色)に沿って動きます。線の外に出ることはできません。

ボタン0(Zキー)を押しながらレバーを操作すると、線の外に出て、新しい線を引くことができます。新しい線が古い線に交わると、囲んだ領域が塗りつぶされます。塗りつぶされるのは、敵がいない方の領域です。

ステージには敵が動き回っています。このサンプルでは、敵がカーソルや線に触れてもミスにはなりません。敵は線や塗りつぶされた領域を避けながら動きます。

ENCLOSED AREA → p. 387

囲んだ領域を塗りつぶす

線で囲んだ領域を塗りつぶすアクションです。「線で囲む」(→p. 174)で解説したように、カーソルが古い線に到達すると、囲んだ領域を塗りつぶすことができます。

新しい線を引くことによって、ステージは2つの領域に分かれます(Fig. 3-59)。塗りつぶされるのは、敵がいない方の領域です。

Fig. 3-59 領域を塗りつぶす

敵がいる方の領域は
塗りつぶさない
新しい線によって
ステージは2つの領域に
分かれる
敵がいない方の領域を
塗りつぶす

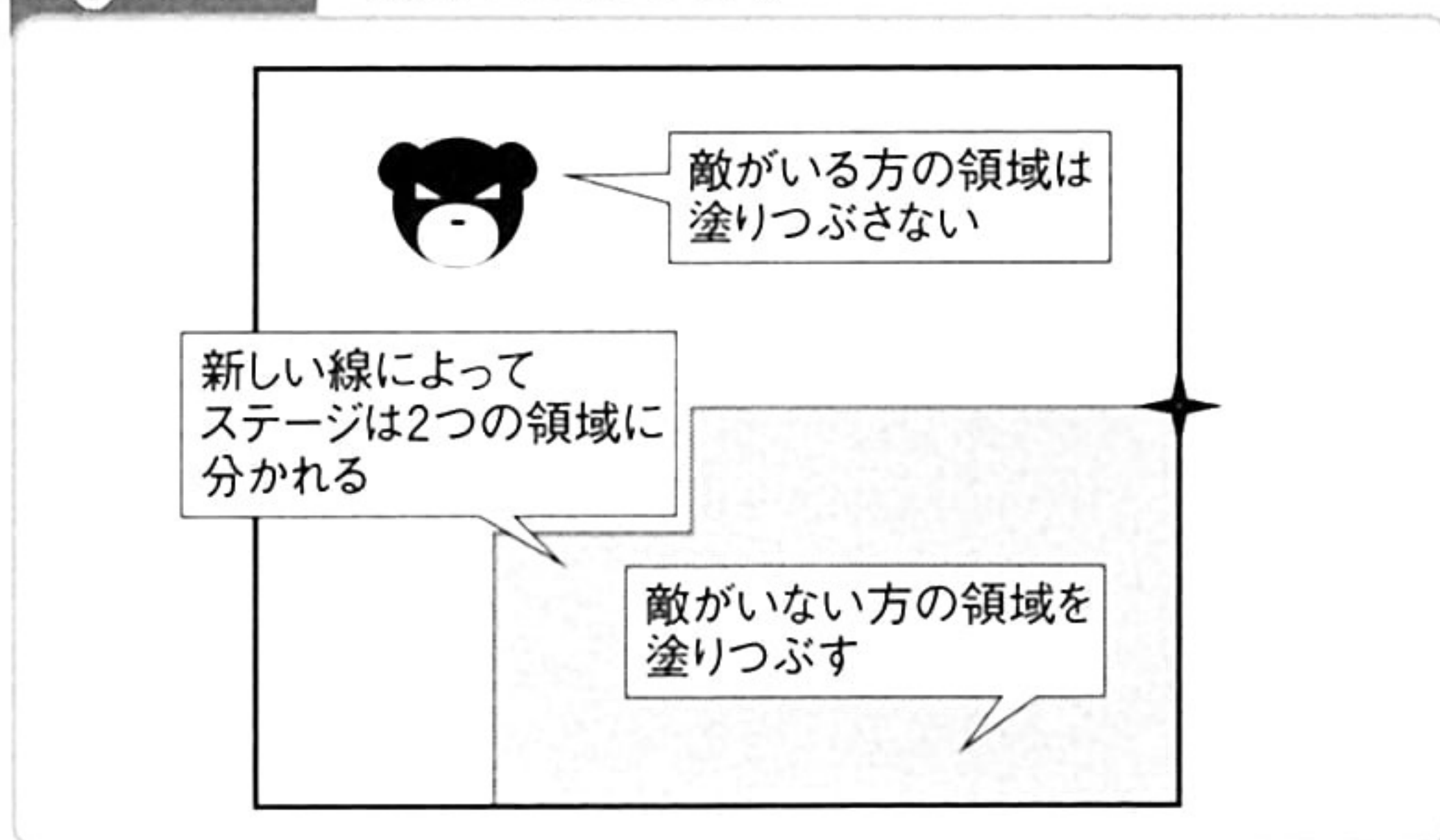
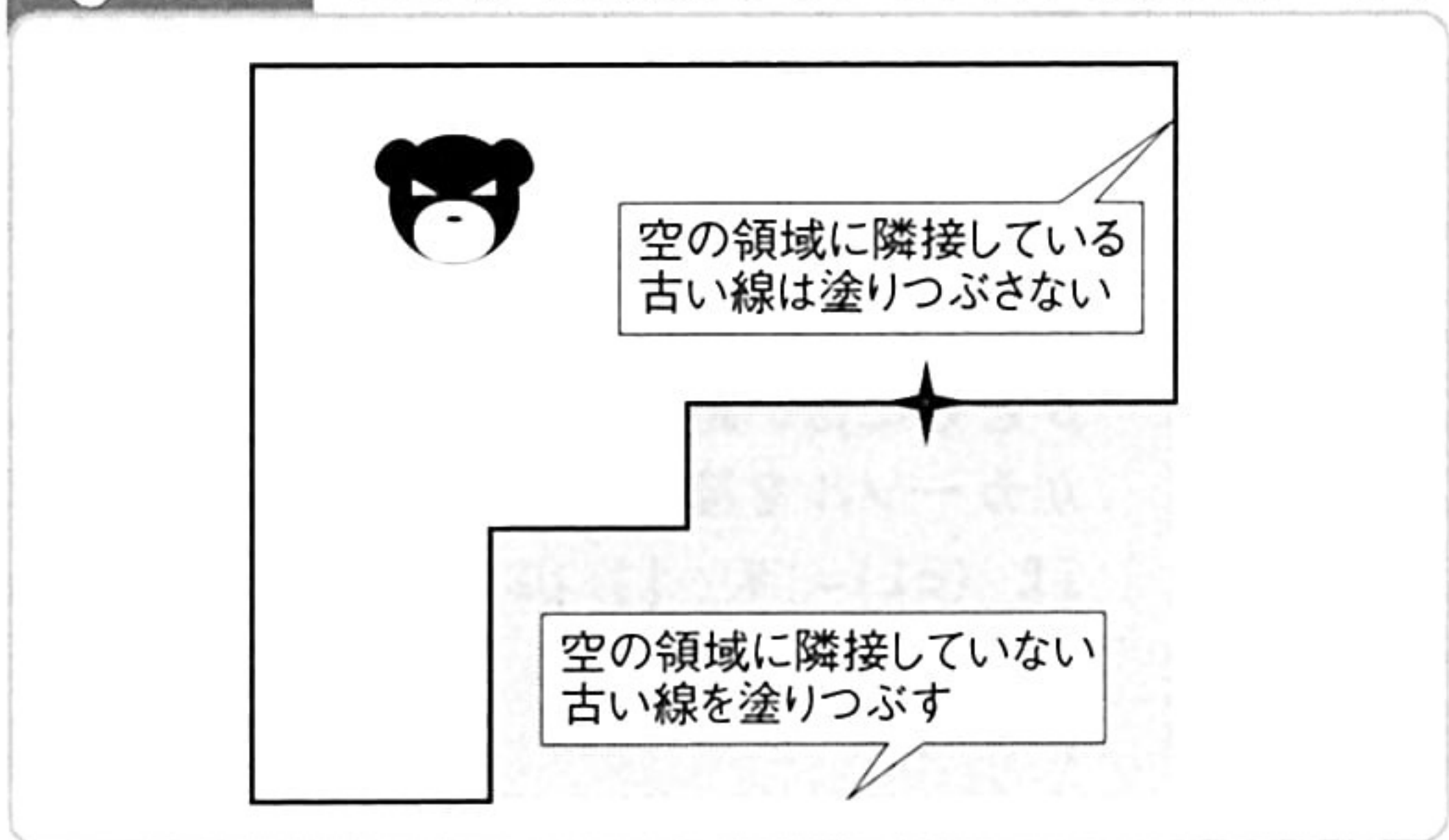



Fig. 3-60 空の領域に隣接していない古い線を塗りつぶす

空の領域に隣接している
古い線は塗りつぶさない
空の領域に隣接していない
古い線を塗りつぶす

領域を塗りつぶすことによって、空の領域に隣接していない古い線ができたときには、その線も塗りつぶします（Fig. 3-60）。これで線が整理され、空の領域の周囲だけに古い線が残ることになります。カーソルは古い線に沿って動くため、これから新しい線を引くことができる空の領域の周囲だけに古い線を限定した方が、プレイしやすくなります。

領域がどんなに複雑な形をしていても、塗りつぶすことができます。敵の動きやステージの形状に応じて、領域の形を工夫しながら攻略することが、このタイプのゲームの面白さです。

アルゴリズム

囲んだ領域を塗りつぶすには、ステージのセルを調べて、隣接する空のセルを次々に塗りつぶしていきます。その際には、複雑な領域でも完全に塗りつぶすことができるように、再帰的な処理を使います。

ステージ内のセルを1つ選び、ここから塗りつぶしを開始します（Fig. 3-61）。どこから塗りつぶしを開始してもよいのですが、後述するように、本書では敵の座標から塗りつぶしを開始することにしています。

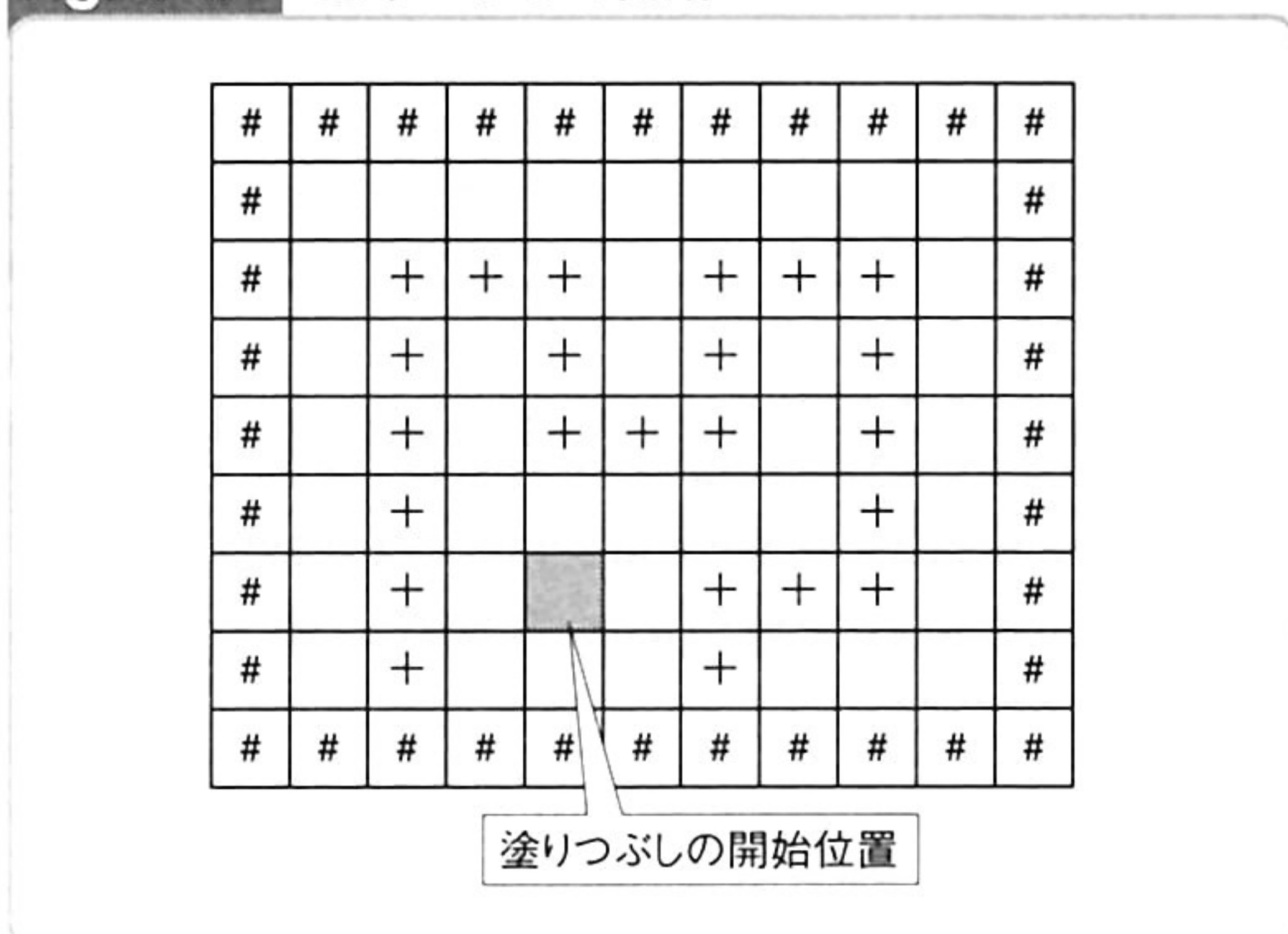
開始セルから左方向へ、空のセルがどこまで続くのかを調べます（Fig. 3-62）。同様に右方向

Fig. 3-61 塗りつぶしの開始

#	#	#	#	#	#	#	#	#	#	#
#										#
#		+	+	+		+	+	+		#
#		+		+		+		+		#
#		+		+	+	+		+		#
#		+						+		#
#		+		(開始)		+	+	+		#
#		+				+				#
#	#	#	#	#	#	#	#	#	#	#

塗りつぶしの開始位置

Fig. 3-62 空のセルの範囲を調べる

#	#	#	#	#	#	#	#	#	#	#
#										#
#		+	+	+		+	+	+		#
#		+		+		+		+		#
#		+		+	+	+		+		#
#		+						+		#
#		+	(空)	(空)	(空)	+	+	+		#
#		+				+				#
#	#	#	#	#	#	#	#	#	#	#

左右に空のセルが
どこまで続くのかを調べる

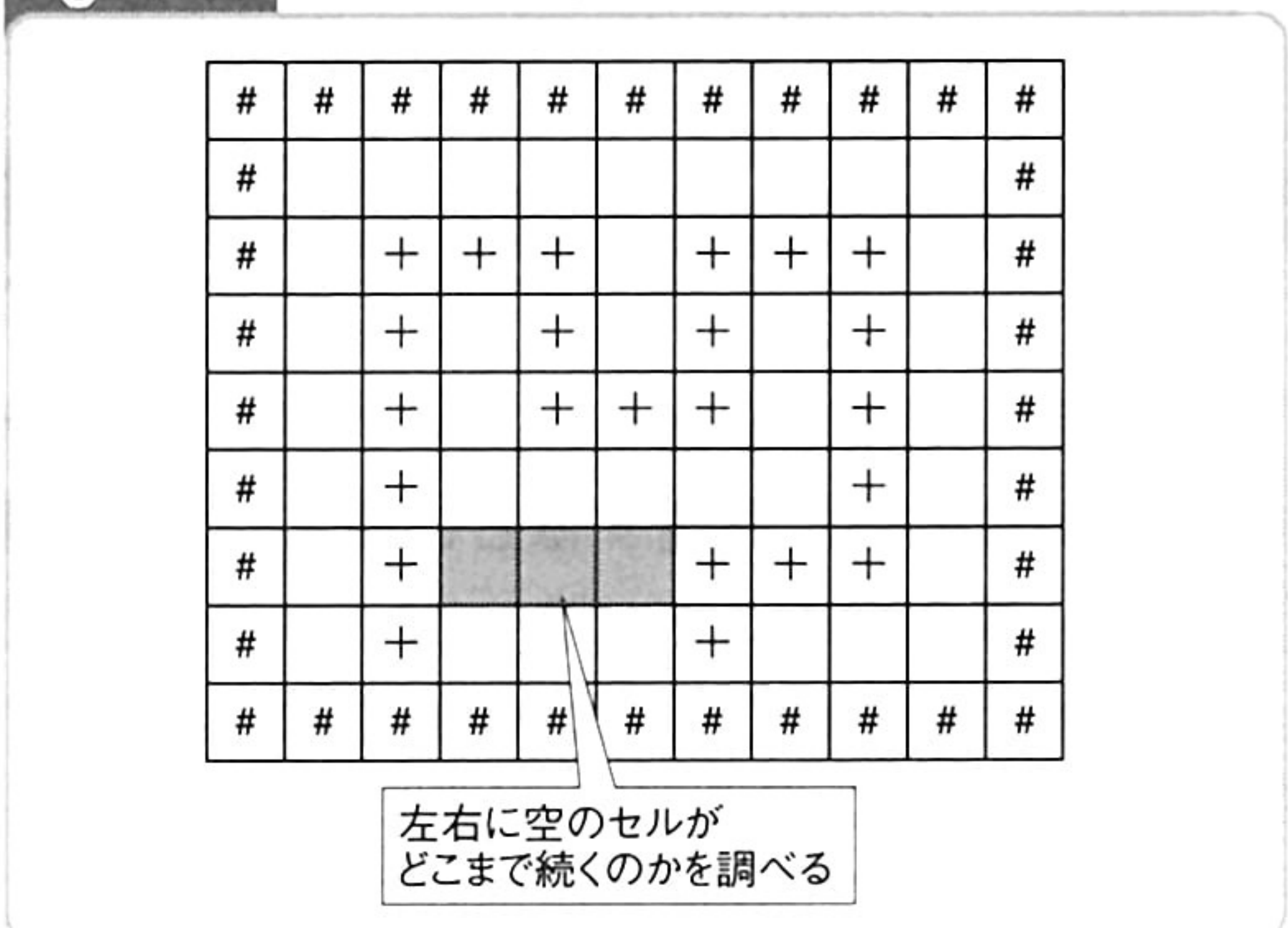



Fig. 3-63 空のセルを塗りつぶす

空のセルの範囲を塗りつぶす

Fig. 3-64 上下のセルを塗りつぶす

上のセルについても同様に塗りつぶす

Fig. 3-65 再帰的に塗りつぶす

再帰的に塗りつぶしを行う
(数字はセルが塗りつぶされる順番)

Fig. 3-66 塗りつぶした領域を反転する

空のセルを
塗りつぶす(*)

塗りつぶした
セル(%)を
空にする

新しい線(+)を
古い線(#)にする

へも、空のセルがどこまで続くのかを調べます。

空のセルの範囲がわかったら、その範囲を塗りつぶします(Fig. 3-63)。ここではセルを「%」に置き換えて塗りつぶすことにします。

次に、塗りつぶした範囲の上下のセルについても、同様に塗りつぶし処理を行います(Fig. 3-64)。例えば上のセルについて、左右方向に空のセルがどこまで続くのかを調べて、その範囲を塗りつぶします。下のセルについても同じ処理を行います。

上下のセルのさらに上下にあるセルについても、そのまた上下にあるセルについても、同様な塗りつぶし処理を再帰的に行います(Fig. 3-65)。図では、セルが塗りつぶされる順番を数字で示しました。このように、塗りつぶし処理を再帰的に行うことによって、複雑な領域を完全に塗りつぶすことができます。

敵がいる座標から塗りつぶしを開始すると、敵がいる領域を塗りつぶすことができます。敵がいない領域を塗りつぶすには、塗りつぶした領域を反転すればよいでしょう(Fig. 3-66)。「%」のセルを空に、空のセルを「*」にすれば、敵がいない領域を塗りつぶすことができます。同時に、新しい線(+)を古い線(#)に変化させます。

Fig. 3-67 空領域に隣接していない古い線を塗りつぶす

*	*	*	*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*	*	*	*
*	*	#	#	#	*	#	#	#	*	*
*	*	#		#	*	#		#	*	*
*	*	#		#	#	#		#	*	*
*	*	#						#	*	*
*	*	#				#	#	#	*	*
*	*	#				#	*	*	*	*
*	*	#	#	#	#	#	*	*	*	*

最後に、空き領域に隣接していない古い線を、塗りつぶします（Fig. 3-67）。古い線の周囲8方向にあるセルについて、空のセルの個数を調べます。空のセルが0個の場合には、その線を塗りつぶします。

プログラム

List 3-7は囲んだ領域を塗りつぶすプログラムです。カーソルの移動処理と、塗りつぶしの処理を掲載しています。

カーソルの移動処理（Move関数）では、カーソルを動かし、新しい線を引きます。カーソルが古い線に到達して、領域を閉じたときには、囲んだ領域を塗りつぶします。敵の座標を開始点として、塗りつぶしの処理を呼び出します。

塗りつぶしの処理（Paint関数）では、左右方向に空のセルが続く範囲を調べ、その範囲を「%」で塗りつぶします。次に、塗りつぶした範囲の上下のセルについて、塗りつぶしの処理を再帰的に呼び出します。

敵がいる領域の塗りつぶしが終わると、移動処理（Move関数）に戻ります。ここでは、敵がいない領域を塗りつぶすために、塗りつぶした領域を反転させます。同時に、新しい線を古い線にしたり、周囲に空領域がない古い線を塗りつぶしたり、といった処理も行います。

塗りつぶした領域の描画については、少し工夫の余地があります。本書のサンプルでは、ステージのセルを調べて、それぞれのセルに対応するグラフィックを描画しています。セルの数が多いと、この処理はかなり負荷が重くなります。サンプルでは、高速化のために、同じ種類のセルをまとめて描画する工夫をしています。詳細は付録CD-ROMの「Puzzle￥Stage3.h」で、CEnclosedAreaStageクラスのDraw関数を参照してください。

より高速に描画するには、塗りつぶした領域を記録するためのテクスチャを用意します。そして、領域を塗りつぶすたびに、直接テクスチャへ塗りつぶした領域を書き込みます。画面に表示するときには、この1枚のテクスチャを描画するだけなので、非常に高速です。テクスチャを扱うコードは、DirectXなどのライブラリに強く依存してしまうので、本書ではこの方法は採用しませんでした。機会があれば、ぜひ実装してみてください。

List 3-7　囲んだ領域を塗りつぶす(CEnclosedAreaCursorクラス)

```
// カーソルの移動処理
bool CEnclosedAreaCursor::Move(const CInputState* is) {

    // ...(中略)...

    // ボタンを押しているときの処理
    if (is->Button[0]) {

        // ...(中略)...

        // 1つ先のセルが空で、
        // 2つ先のセルが元からある線のときには、
        // 領域を閉じる
        if (c1==' ' && c2=='#') {

            // 1つ先のセルを線にする
            Cell->Set(CX+vx, CY+vy, '+');

            // 敵がいる領域を「%」で塗りつぶす
            Paint(Enemy->CX, Enemy->CY);

            // 領域の塗りつぶしを反転させることによって、
            // 敵がいない領域を塗りつぶす
            // また、新しい線を古い線にする
            for (int y=3; y<Cell->GetYSize()-3; y++) {
                for (int x=3; x<Cell->GetXSize()-3; x++) {
                    switch (Cell->Get(x, y)) {

                        // 敵がいない領域(空のセル)は
                        // 塗りつぶす(*)
                        case ' ':
                            Cell->Set(x, y, '*');
                            break;

                        // 敵がいる領域(%)は空にする
                        case '%':
                            Cell->Set(x, y, ' ');
                            break;

                        // 新しい線(+)は古い線(#)にする
                        case '+':
                            Cell->Set(x, y, '#');
                            break;
                    }
                }
            }

            // 周囲に空き領域がない古い線を塗りつぶす
```

```
            for (int y=2; y<Cell->GetYSize()-2; y++) {
                for (int x=2; x<Cell->GetXSize()-2; x++) {

                    //古い線に対する処理
                    if (Cell->Get(x, y)=='#') {

                        //線の周囲(8方向)にある
                        //空のセルを数える
                        int count=0;
                        for (int i=-1; i<=1; i++) {
                            for (int j=-1; j<=1; j++) {
                                if (Cell->Get(x+i, y+j)==' ') {
                                    count++;
                                }
                            }
                        }

                        //空のセルがまったくないときには、
                        //古い線を塗りつぶす
                        if (count==0) {
                            Cell->Set(x, y, '*');
                        }
                    }
                }
            }
        } else

        //...(中略)...
    } else

    //...(中略)...
}

//塗りつぶしの処理
void CEnclosedAreaCursor::Paint(int x, int y) {

    //現在のセルが空ならば、
    //塗りつぶしを行う
    if (Cell->Get(x, y)==' ') {
        int lx, rx;

        //空のセルが続くかぎり、
        // 現在のセルから左へ移動する
        for (lx=x; Cell->Get(lx-1, y)==' '; lx--) ;

        //空のセルが続くかぎり、
        //現在のセルから右へ移動する
        for (rx=x; Cell->Get(rx+1, y)==' '; rx++) ;
```

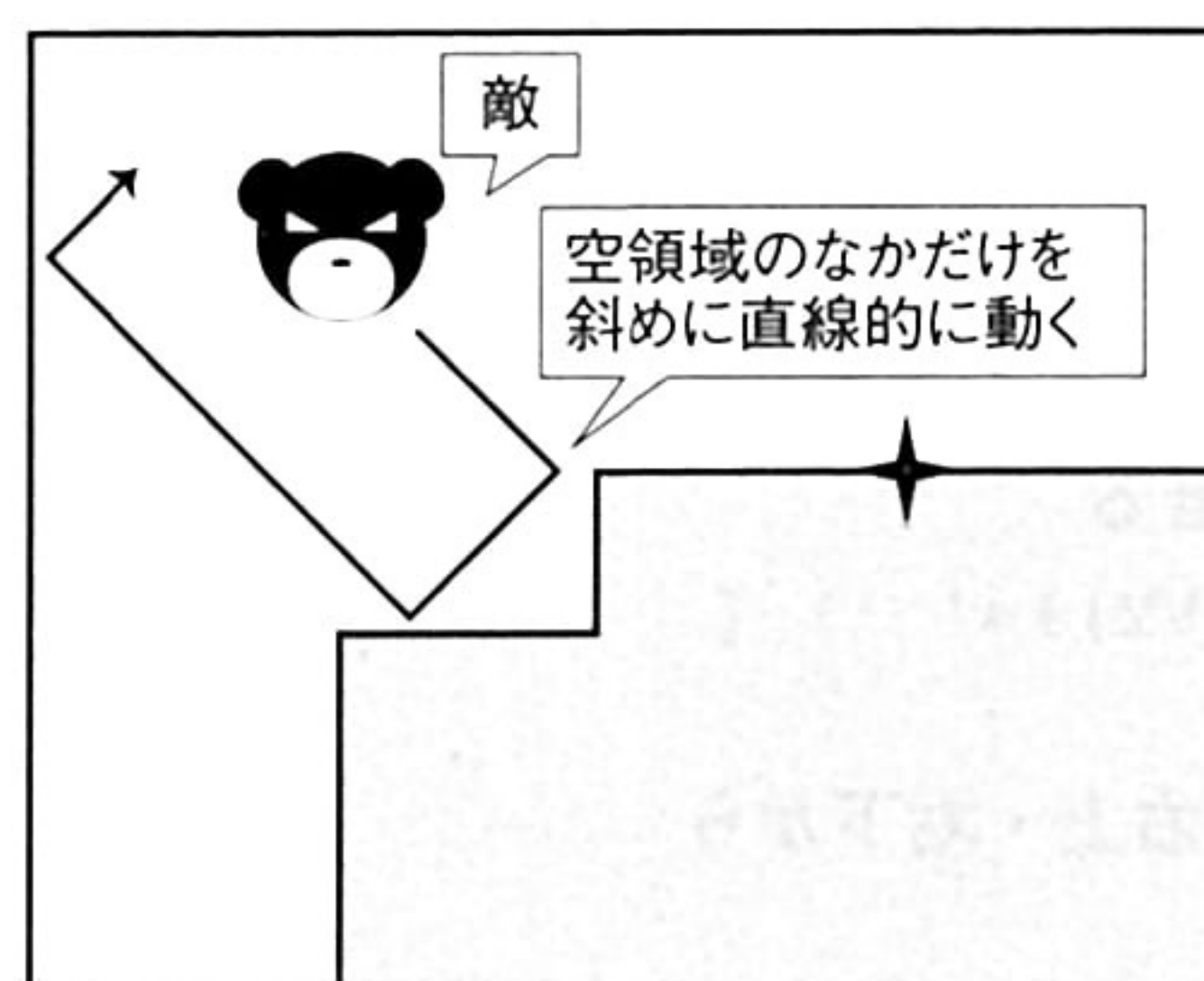
```
            //空のセルを「%」で塗りつぶす
            for (x=lx; x<=rx; x++) {
                Cell->Set(x, y, '%');
            }

            //塗りつぶしたセルの上下のセルについて、
            //塗りつぶしの処理を再帰的に行う
            for (x=lx; x<=rx; x++) {
                Paint(x, y-1);
                Paint(x, y+1);
            }
        }
    }
```

囲まれた領域を避けて動く敵

「線で囲む」(→p. 174)の敵です。線で囲まれて塗りつぶされた領域を避けて、空領域だけを動き回ります。

敵の動きはゲームによっていろいろですが、本書のサンプルでは、斜め方向に直線的に動く敵にしました(Fig. 3-68)。敵は空領域のなかだけを動きます。線を乗り越えたり、塗りつぶされた領域に入ったりすることはありません。

ゲームによっては、空領域を動く敵の他に、線に沿って進む敵が出現することもあります。また、サンプルでは敵がカーソルに触れてもミスになりませんが、敵がカーソルに触れてはいけないゲームや、敵が線にさえ触れてはいけないゲームもあります。

Fig. 3-68 空領域を動き回る敵

アルゴリズム

囲まれた領域を避けて動く敵を実現するには、敵を動かすたびに、ステージのセルを調べます。敵の移動先のセルを調べて、セルが空のときだけ、敵を移動させます(Fig. 3-69)。セルが空ではないとき、つまり線や塗りつぶされた領域のときには、敵の移動方向を変えます(Fig. 3-70)。これで、敵は空領域のなかだけを動くようになります。

Fig. 3-69 セルが空ならば移動させる

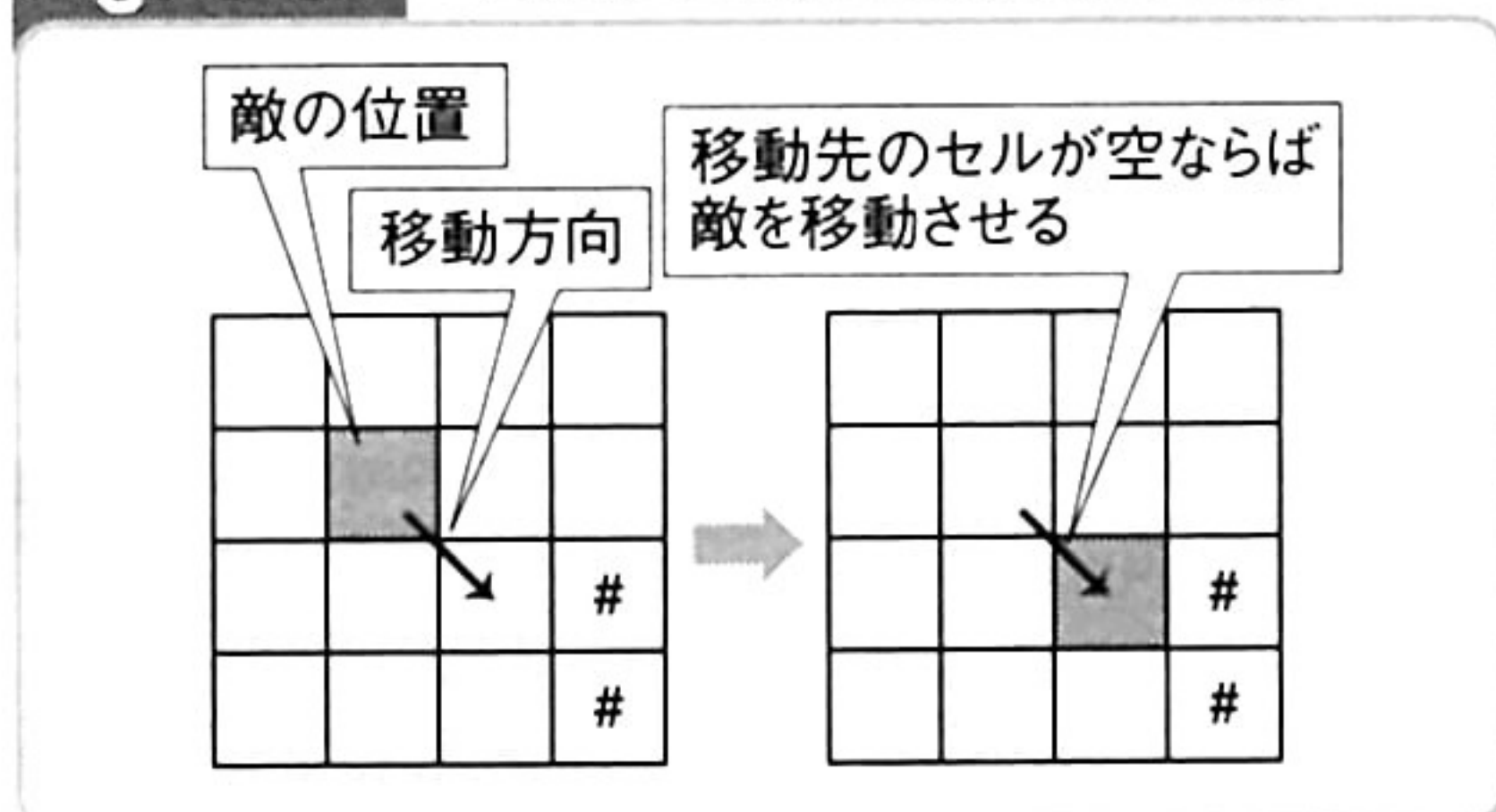
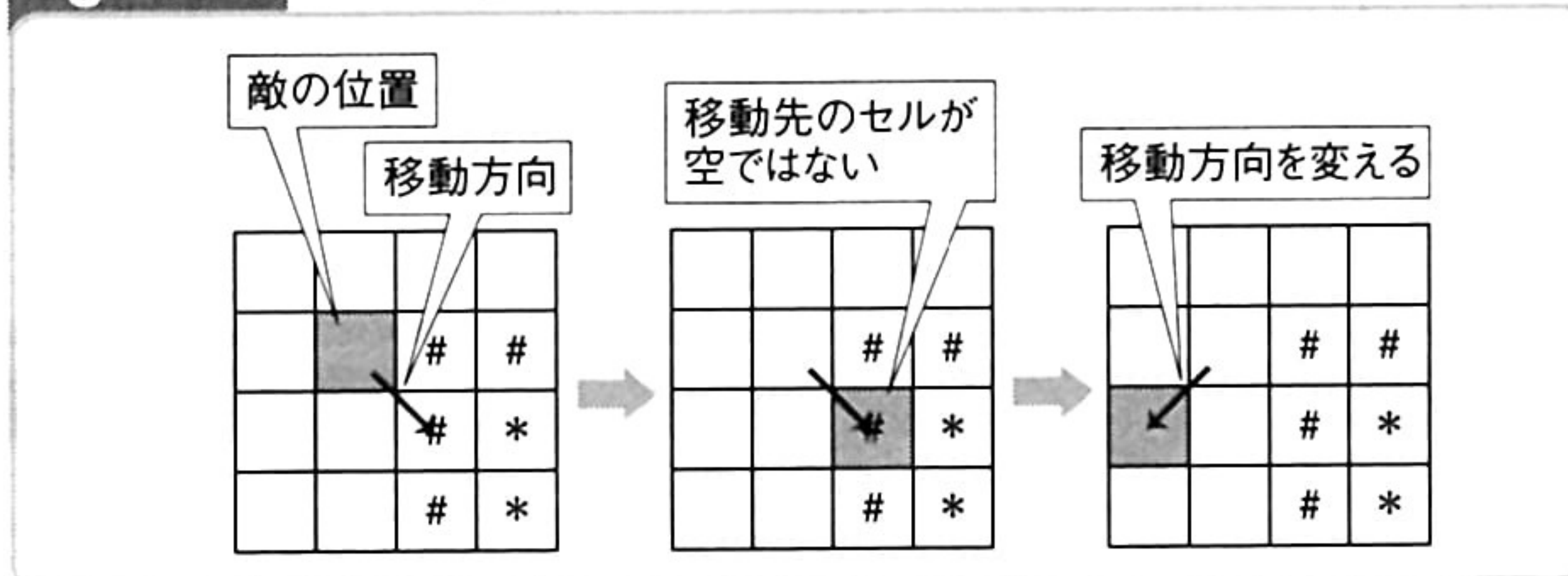



Fig. 3-70 セルが空でなければ移動方向を変える

プログラム

List 3-8は囲まれた領域を避けて動く敵のプログラムです。敵の移動処理(Move関数)を掲載しました。

敵を動かす前に、移動先のセルを調べます。セルが空ではないときには、移動方向を変化させます。ここでは左上・左下・右上・右下のいずれかを、ランダムに選ぶことにしました。セルが空のときには、敵を直進させます。

List 3-8 囲まれた領域を避けて動く敵(CEnclosedAreaEnemyクラス)

```
// 敵の移動処理
bool CEnclosedAreaEnemy::Move(const CInputState* is) {

    // 移動先のセルを調べて、
    // 空ではない(線または塗りつぶされた領域)ならば、
    // 移動方向をランダムに変化させる
    if (Cell->Get(CX+VX, CY+VY)!=' ') {

        // 移動方向を左上・左下・右上・右下から
        // ランダムに選ぶ
        VX=Rand.Int31()%2*2-1;
        VY=Rand.Int31()%2*2-1;
    } else

    // 移動先のセルが空のときには直進する
```

```
    {
        CX+=VX;
        CY+=VY;
    }

    return true;
}
```

一筆書きでアイテムを回収する

キャラクターを動かし、一筆書きの要領でアイテムを回収するアクションです。キャラクターが一度通った場所は、二度と通ることができません。

迷路状のステージを考えましょう（Fig. 3-71）。ステージにはキャラクターと、複数のアイテムが配置されています。

キャラクターはレバー入力で上下左右に動きます（Fig. 3-72）。通路上を移動することはできますが、壁を通り抜けることはできません。また、キャラクターが動くと軌跡が残りますが、この軌跡も通過することができません。

アイテムにキャラクターを重ねると、アイテムを取ることができます（Fig. 3-73）。ステージ内のアイテムをすべて取ることがゲームの目的ですが、問題はキャラクターの軌跡を通過できないことです（Fig. 3-74）。適切なルートでアイテムを回収しないと、軌跡にジャマされて、キャラクターを動かせなくなってしまいます。同じ道を二度以上通ることなく、すべてのアイテムを一筆書きの要領で集められるようなルートを、よく考えてからキャラクターを動かす必要があります。

一筆書きでアイテムを回収するゲームには、『チェックマン』などがあります。このゲームでは、キャラクターを動かして、敵や障害物に接触しないように、アイテムを回収します。キ

Fig. 3-71 アイテムが配置されたステージ

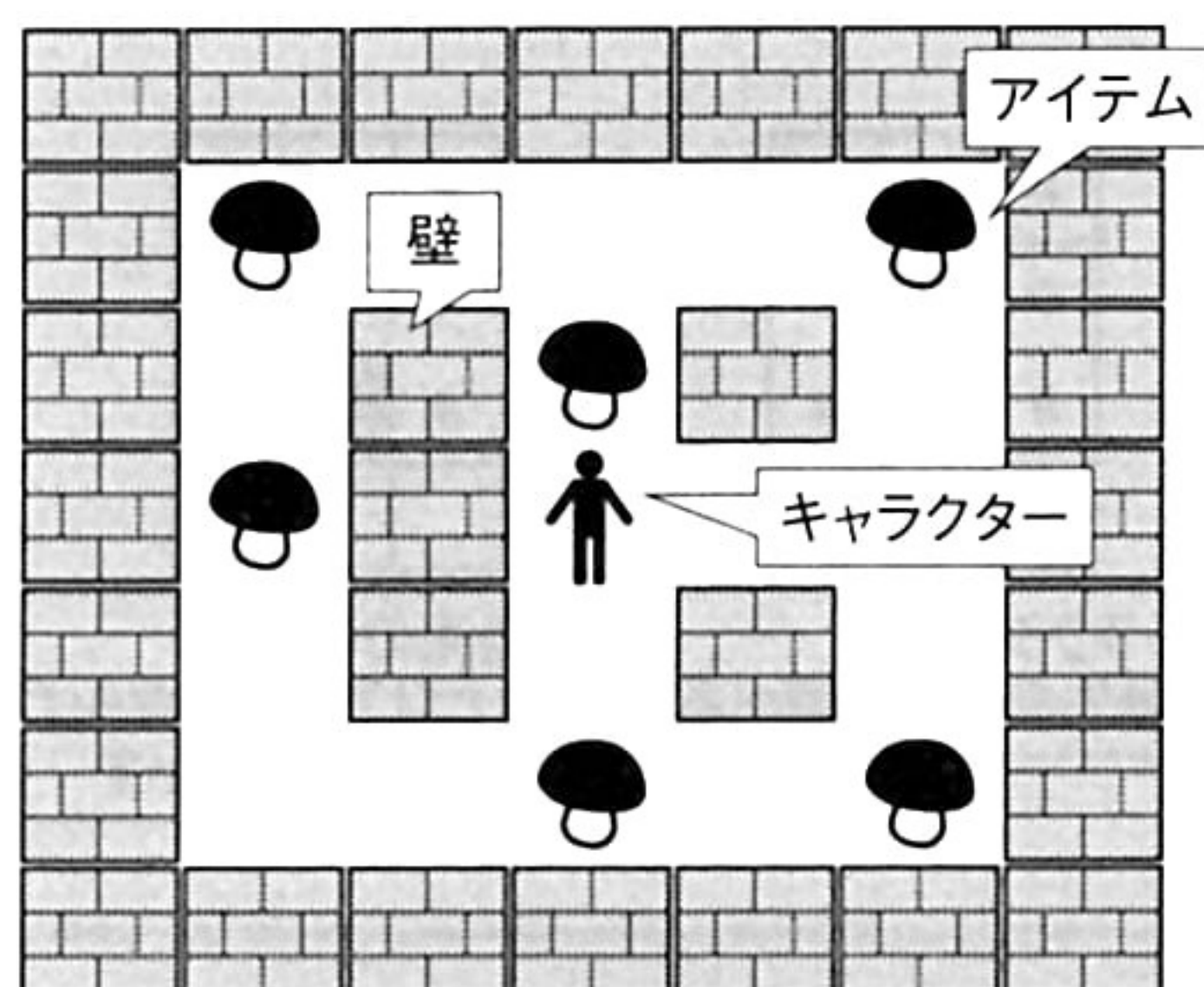



Fig. 3-72 キャラクターを動かす

Fig. 3-73 アイテムを取る

アイテムに重なると
取ることができる
ステージ内のアイテムを
すべて取ることが目的

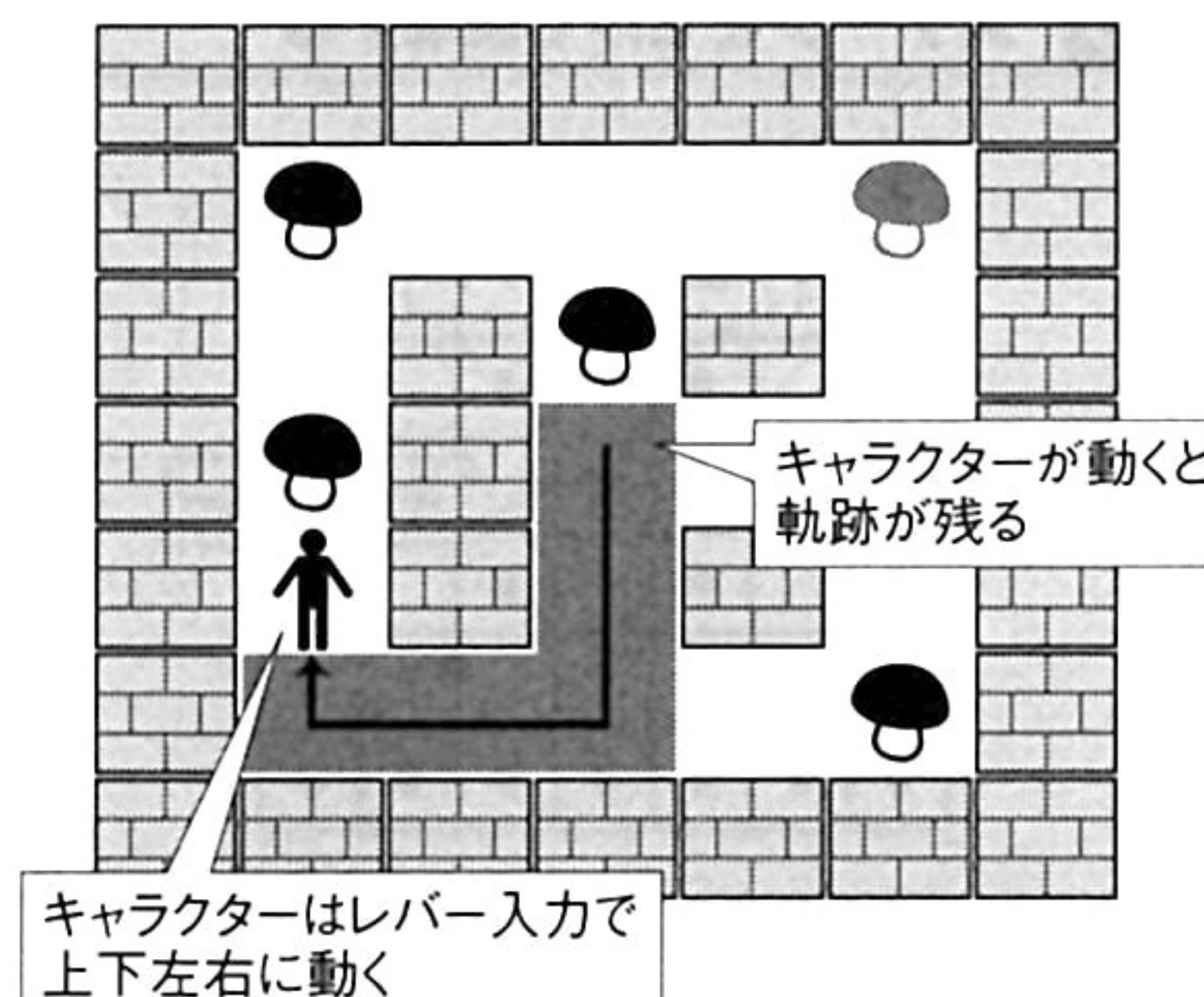
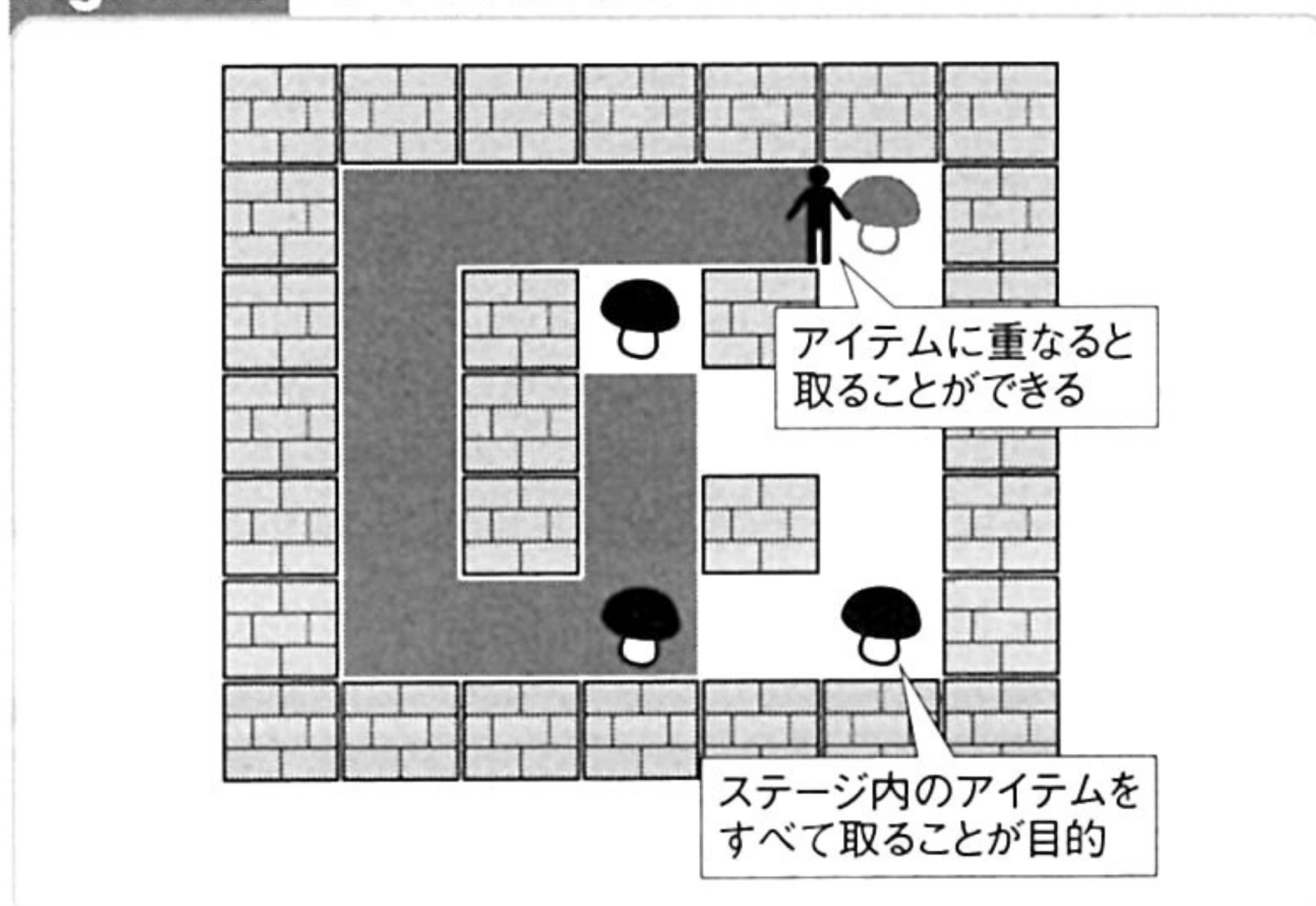
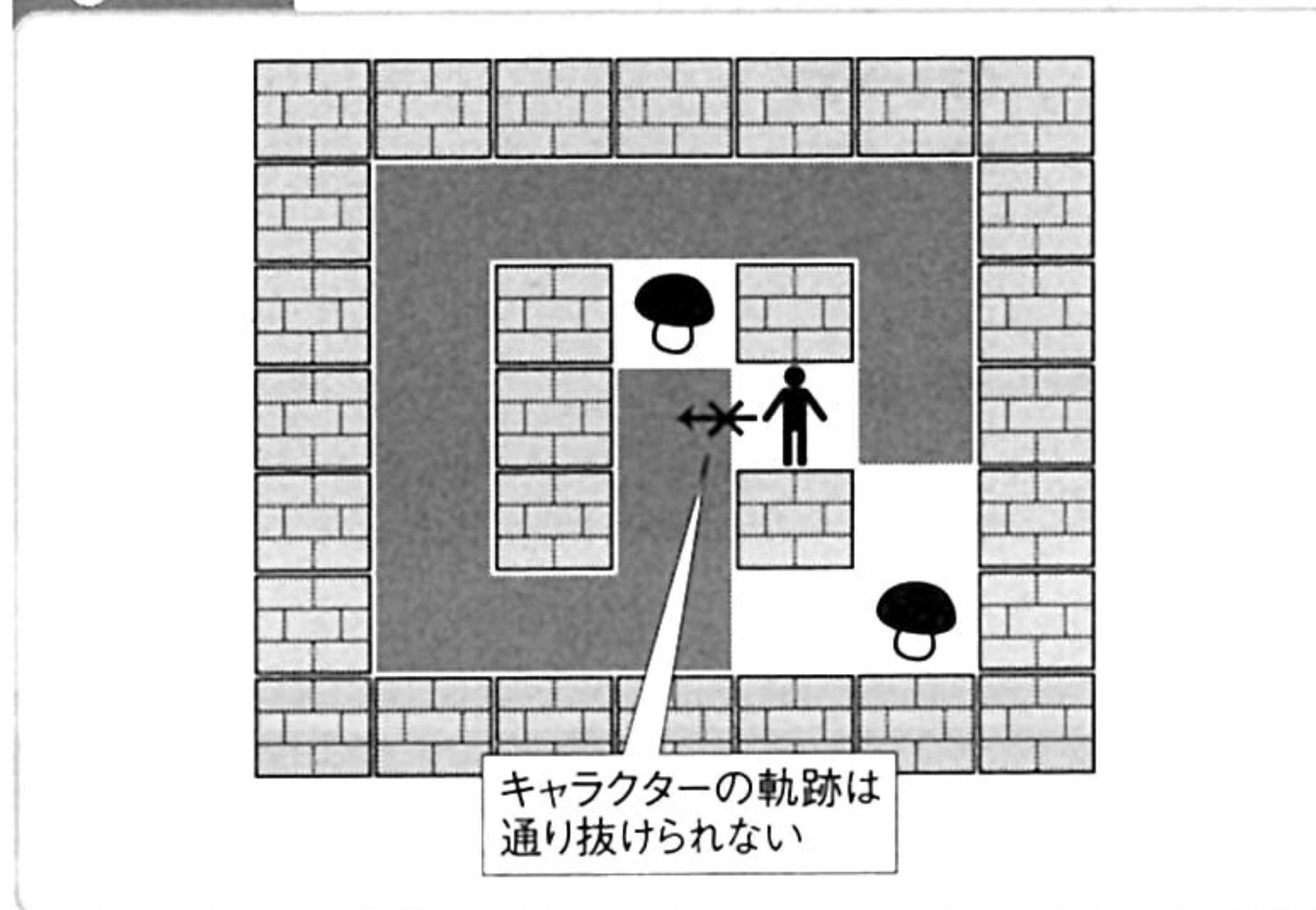



Fig. 3-74 キャラクターの軌跡は通過できない

キャラクターの軌跡は
通り抜けられない

ャラクターが動くと軌跡が残り、この軌跡を再び通過することはできないため、一筆書きの要領でアイテムを集めることになります。画面の両端がつながってループしているので、取れそうにないアイテムでも、意外な回収ルートが見つかることがあります。

アルゴリズム

一筆書きでアイテムを回収するアクションは、「迷路を歩く」(→p. 10) の応用で実現することができます。まず、ステージをセルで表現します (Fig. 3-75)。壁は「#」、アイテムは「I」、キャラクターは「M」で表しました。

レバー入力があったら、キャラクターを上下左右に動かします。キャラクターを動かす前には、移動先のセルを調べます。セルが空またはアイテムのときには、キャラクターを移動させることができます (Fig. 3-76)。セルが壁または軌跡のときには、キャラクターを移動させることはできません (Fig. 3-77)。

キャラクターを移動させたら、キャラクターの軌跡をセルに書き込みます (Fig. 3-78)。軌跡は「+」で表しました。キャラクターが動くにつれて、軌跡が伸びていき、通れない場所が増えていきます。

Fig. 3-75 ステージをセルで表現する

#	#	#	#	#	#	#
#	I					#
#		#	I	#		#
#	I	#	M			#
#		#		#		#
#			I		I	#
#	#	#	#	#	#	#

壁
アイテム
キャラクター

Fig. 3-76 キャラクターが移動できる場合

キャラクターの
移動先のセルを調べる
移動先のセルが
空またはアイテムならば
キャラクターを動かす

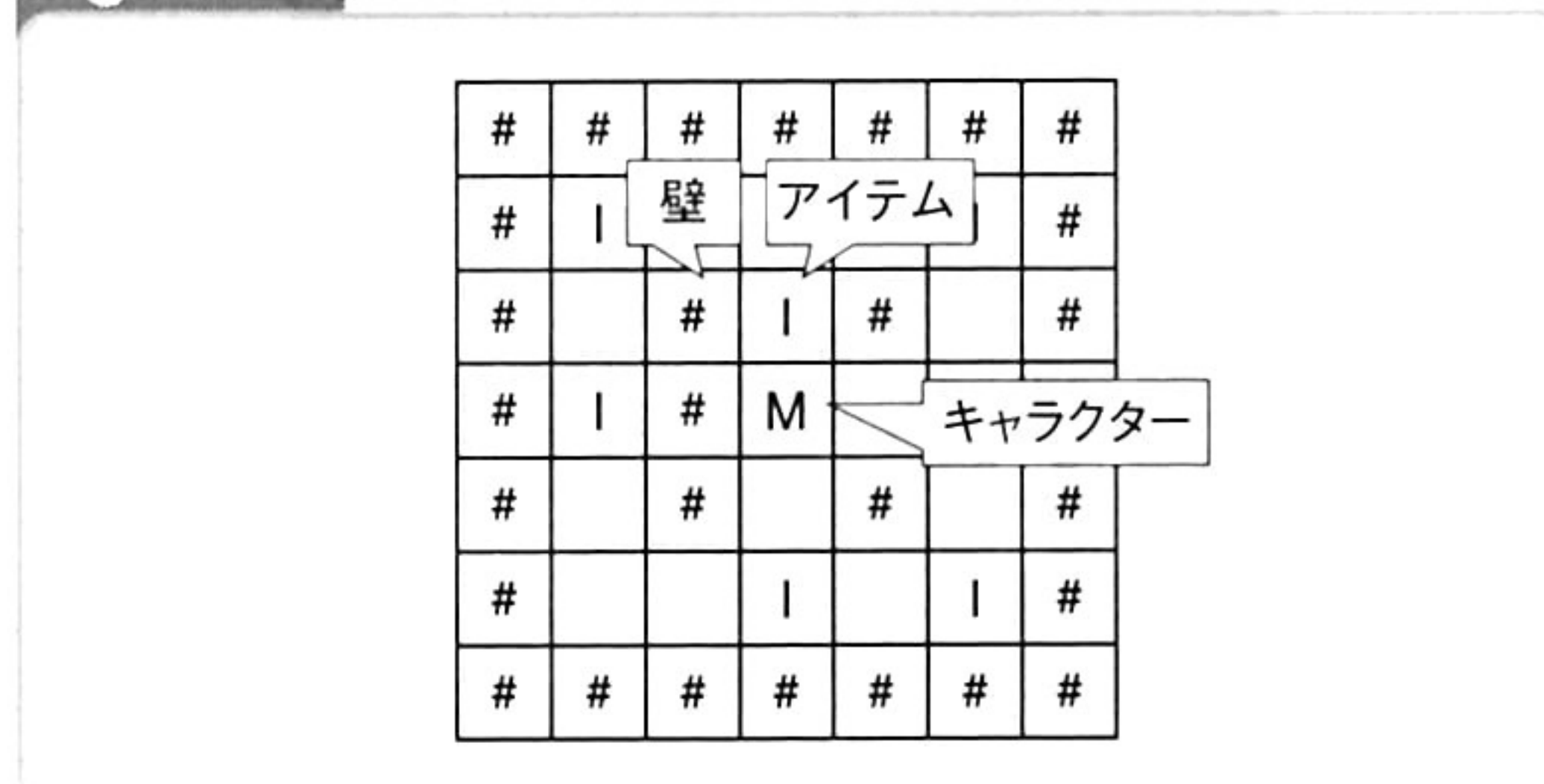
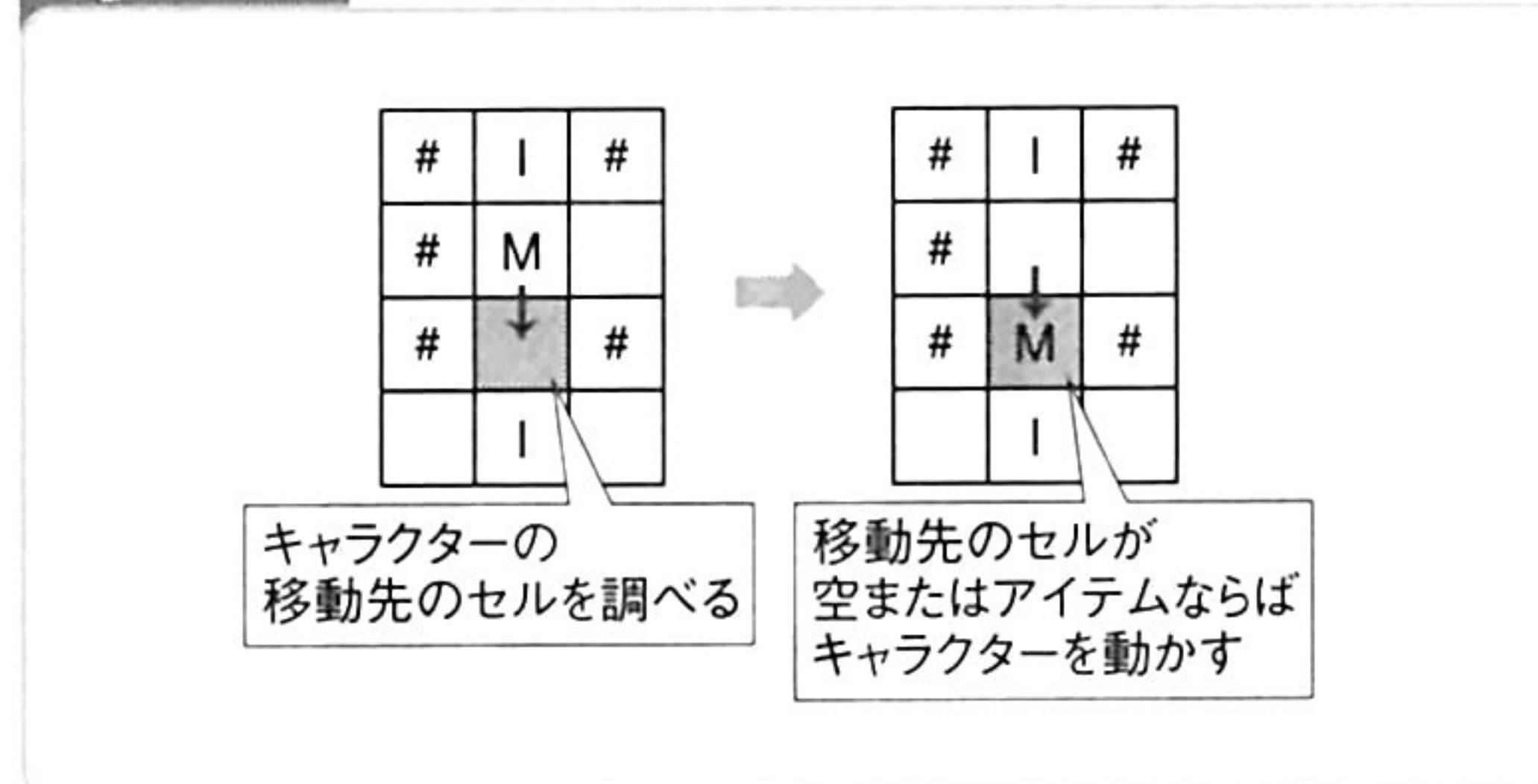



Fig. 3-77 キャラクターが移動できない場合

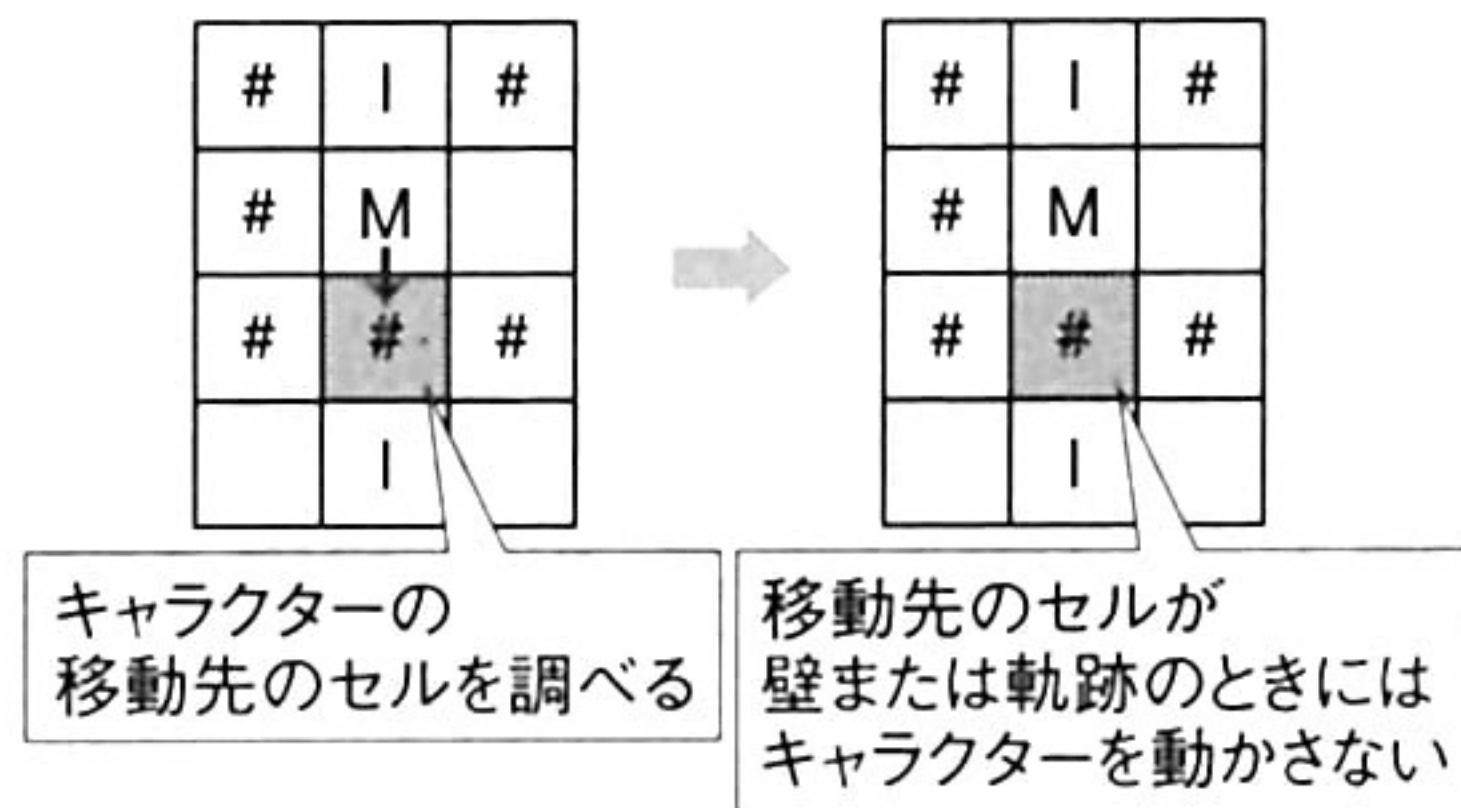
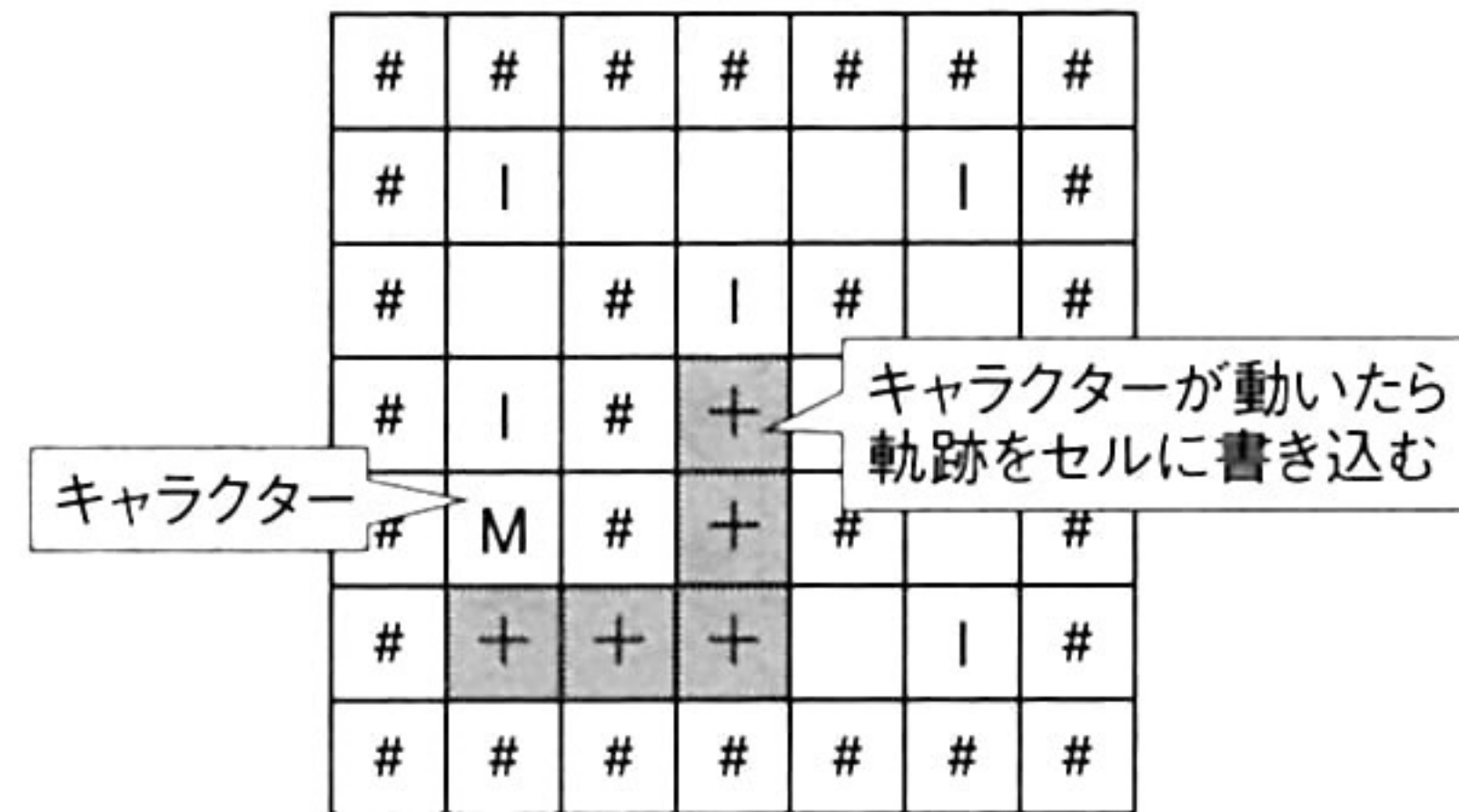
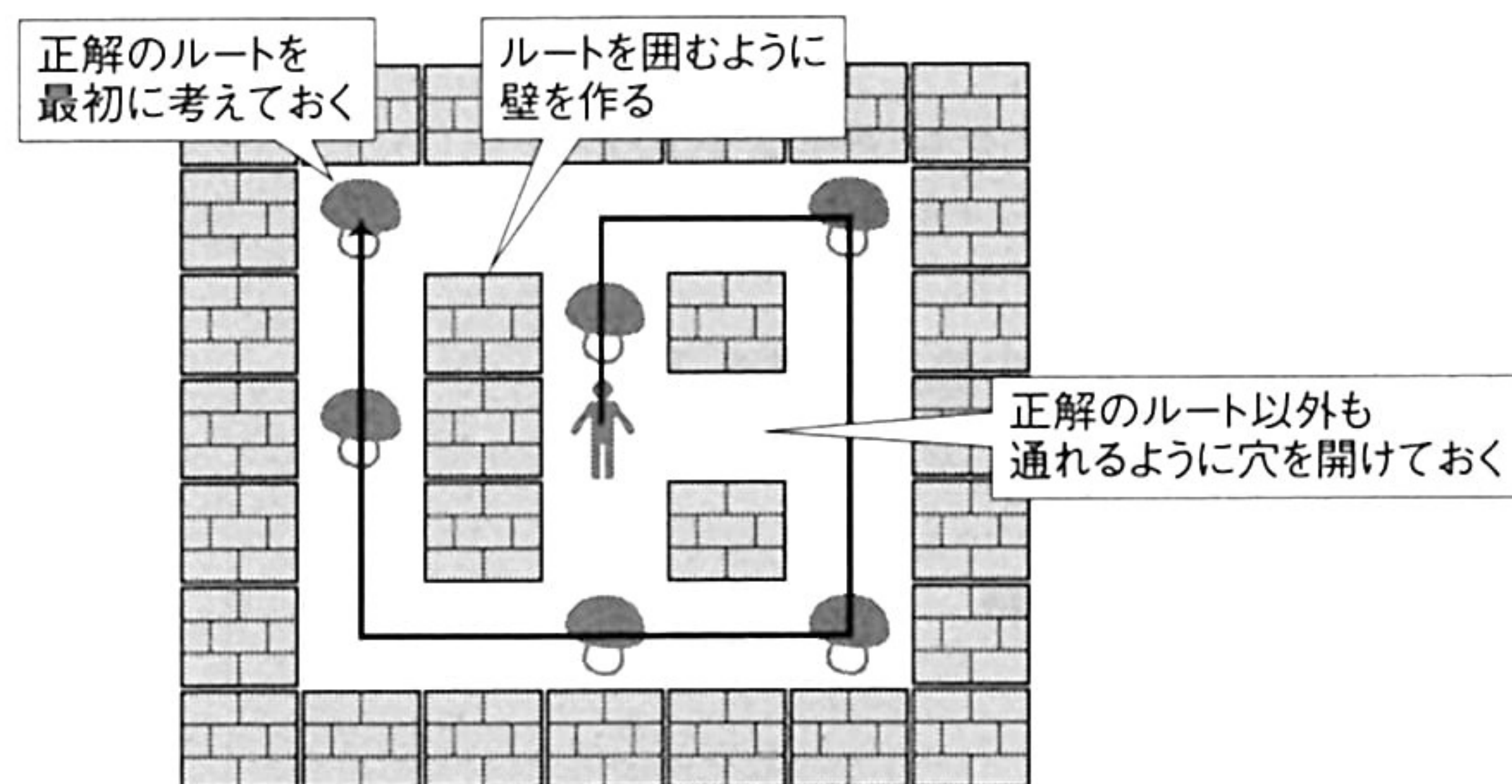



Fig. 3-78 軌跡を書き込む

ステージ作成のポイント

一筆書きでアイテムを回収するゲームのポイントは、ステージの作成方法です。アイテムを一筆書きで回収できる配置にするには、あらかじめ一筆書きのルートを考えておき、そのルートを囲むように壁を作るのが簡単です（Fig. 3-79）。また、正解のルート以外も通れるように、壁のところどころに穴を開けておくと、問題を難しくすることができます。この方法を応用することで、ステージを自動生成することも不可能ではないでしょう。

一方、『チェックマン』のように、ステージを迷路状にしない方法もあります。このゲームでは、ステージに障害物はありますが、迷路というほどではありません。また、アイテムは時間とともにランダムな位置に出現するので、あらかじめルートを完全に考えてから動くというスタイルでもありません。むしろ、自分の軌跡にジャマされつつ、アイテムへのルートを考えたり、敵をかわしたりといったところに、このゲームの面白さがあります。

Fig. 3-79 軌跡を書き込む

プログラム

List 3-9は一筆書きでアイテムを回収するプログラムです。キャラクターの移動処理を掲載しました。

移動処理では、キャラクターの状態を入力状態・移動状態に分けています。入力状態では、レバー入力に応じて、キャラクターを上下左右に動かします。移動先のセルを調べて、セルが空やアイテムのときだけ、キャラクターを移動させます。回収したアイテムを消すために、キャラクターの移動先のセルは空にします。

移動状態では、キャラクターを画面上で滑らかに動かします。タイマーを使って、キャラクターの描画座標を少しずつ変化させることによって、画面上ではキャラクターが滑らかに動いているように見えます。

List 3-9　一筆書きでアイテムを回収する（CTraversableRouteManクラス）

```
// キャラクターの移動処理
bool CTraversableRouteMan::Move(const CInputState* is) {

    // 入力状態
    if (State==0) {

        // レバーの入力に応じて、
        // 上下左右に移動方向を設定する
        VX=VY=0;
        if (is->Left) VX=-1; else
        if (is->Right) VX=1; else
        if (is->Up) VY=-1; else
        if (is->Down) VY=1;

        // レバーが入力されたときの処理
        if (VX!=0 || VY!=0) {

            // 移動先のセルを調べる
            char c=Cell->Get(CX+VX, CY+VY);

            // 移動先のセルが空またはアイテムのとき
            if (c==' ' || c=='I') {

                // キャラクターの軌跡を書き込む
                Cell->Set(CX, CY, '+');

                // セル座標の更新
                CX+=VX;
                CY+=VY;

                // アイテムを消すために、
                // セルを空にする
                Cell->Set(CX, CY, ' ');

                // タイマーを設定し、
                // 移動状態に移行する
                State=1;
                Time=0;
```

```
            }
        }
    }

    // 移動状態
    if (State==1) {

        // タイマーの更新
        Time++;

        // 画面上でキャラクターが滑らかに動くように、
        // 描画座標を少しずつ変化させる
        X=CX-VX*(1-Time*0.1f);
        Y=CY-VY*(1-Time*0.1f);

        // 一定時間が経過したら、入力状態に移行する
        if (Time==10) {
            State=0;
        }
    }

    return true;
}
```

SAMPLE

「TRAVERSABLE ROUTE」は「一筆書きでアイテムを回収する」のサンプルです。レバーの上下左右（カーソルキーの上下左右）でキャラクターが移動します。通路やアイテムは通れますが、壁や軌跡を通過することはできません。

キャラクターが動くたびに、軌跡が伸びていきます。自分の軌跡に進路を阻まれないように、一筆書きの要領ですべてのアイテムを回収することが目的です。

TRAVERSABLE ROUTE → p. 387

言葉を作る

文字を縦横に並べて、さまざまな言葉を作るアクションです。見た目はクロスワードに似ていますが、解答が一通りに決まっているクロスワードとは違い、どんな言葉を作ってもかまわないので、ゲーム性はかなり異なります。

ステージには文字を格子状に並べることができます（Fig. 3-80）。本書のサンプルは、ステージにまったく文字がない状態から始まりますが、ゲームによってはあらかじめいくつかの文字が並んでいる場合もあります。

Fig. 3-80 文字を格子状に並べる

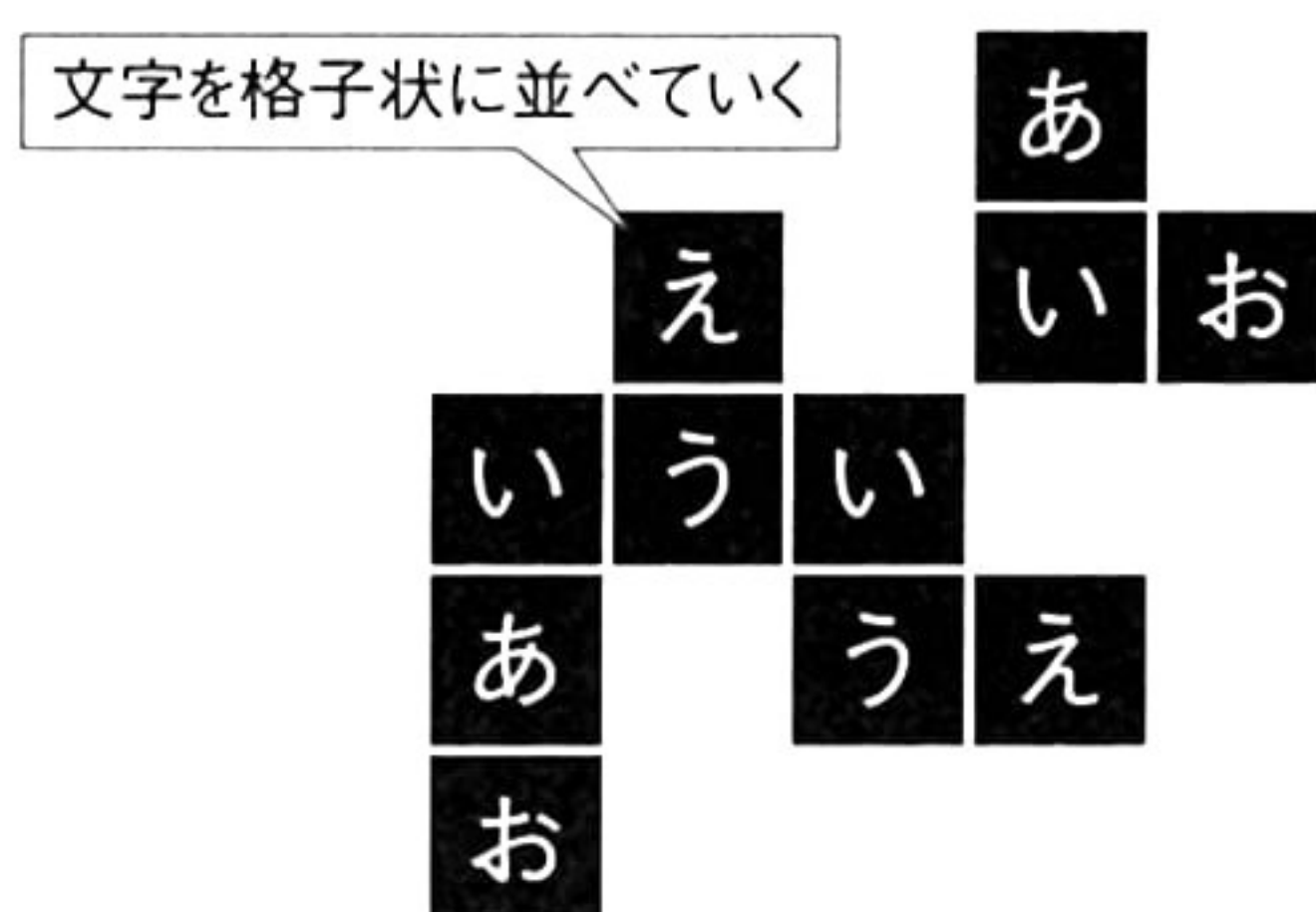



Fig. 3-81 新しい文字を置く

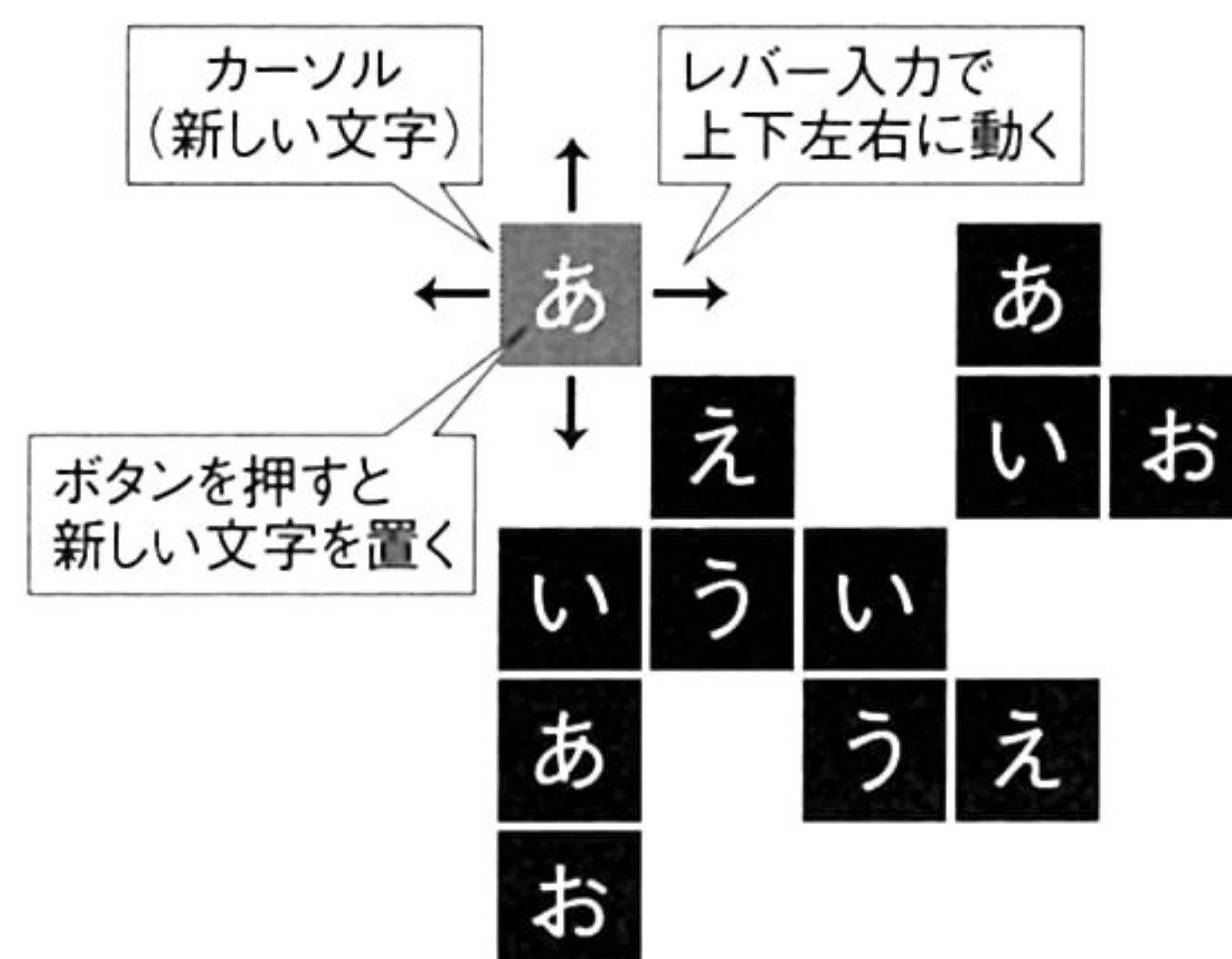



Fig. 3-82 言葉ができる

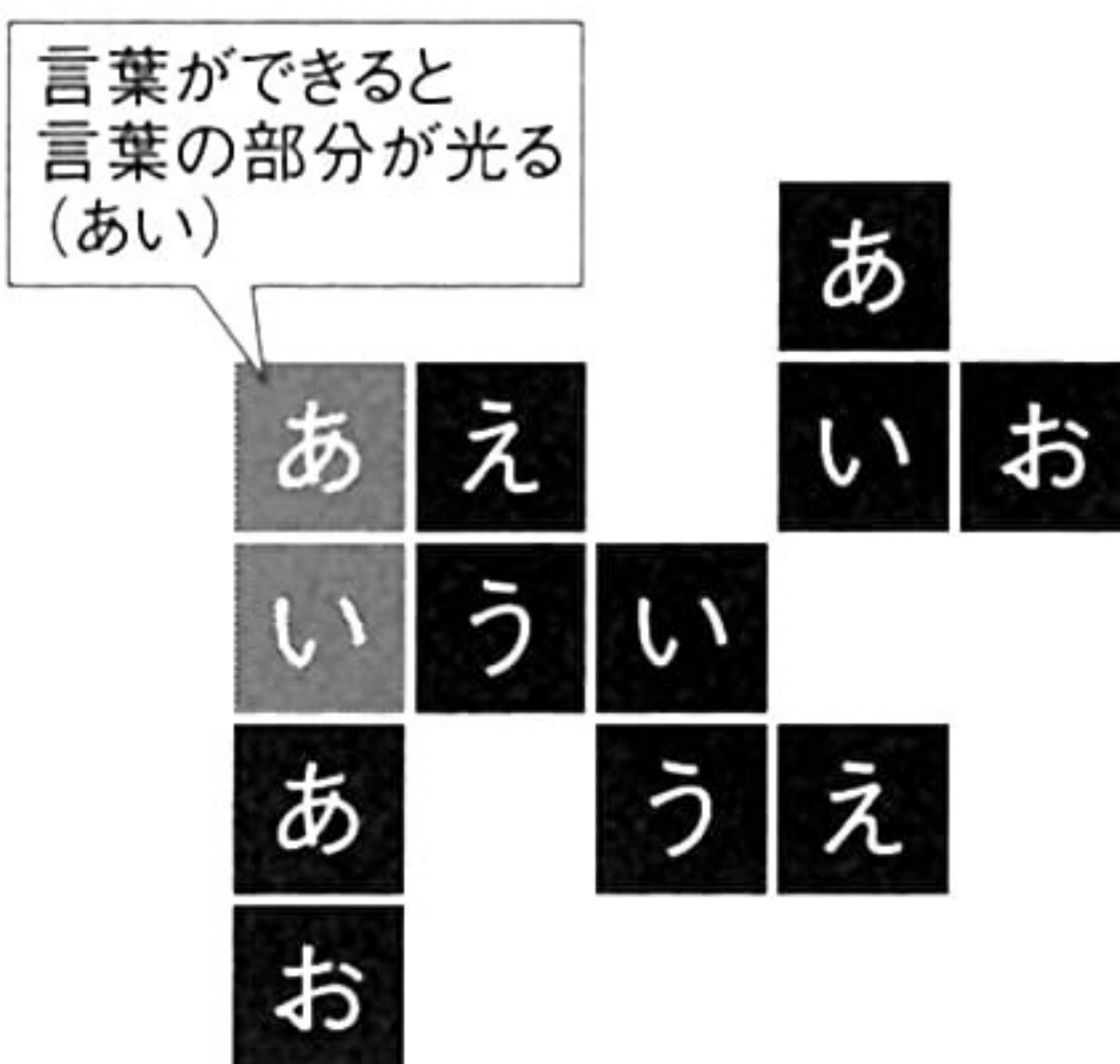
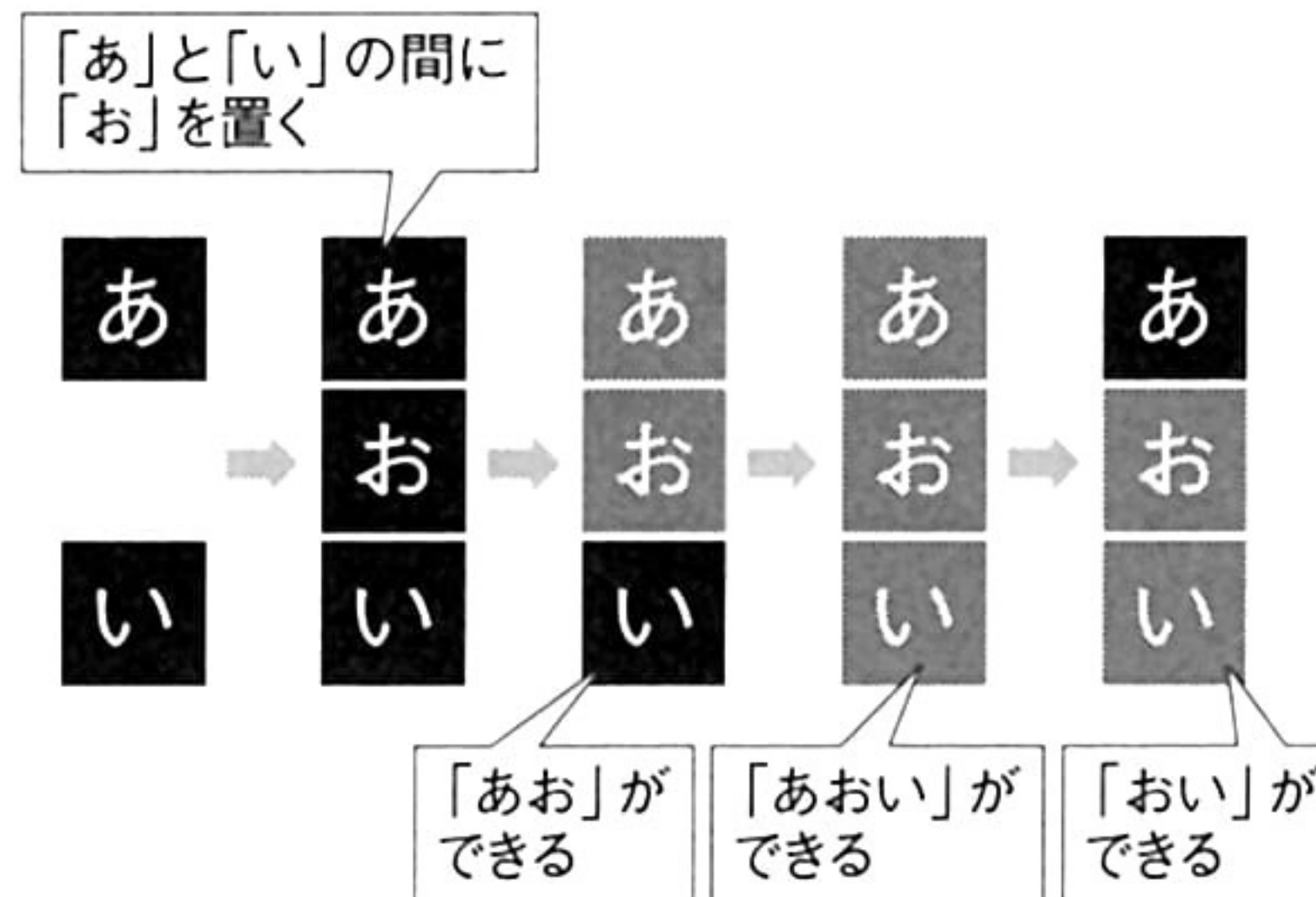



Fig. 3-83 複数の言葉を同時に作る

新しい文字を置くには、カーソルを上下左右に動かして、文字を置く場所を決めます（Fig. 3-81）。ボタンを押すと、その場所に新しい文字を置くことができます。本書のサンプルでは、古い文字（すでに置かれている文字）を黒色で、新しい文字（カーソル）を灰色で示しています。新しい文字の種類はランダムに決まります。ゲームによっては、置く文字をプレイヤーが選べる場合もあります。

文字を置いたときに、周囲の文字と組み合わせて言葉ができていると、言葉の部分が光ります（Fig. 3-82）。これは、ゲームが言葉の辞書を持っていて、この辞書のなかにある言葉のいずれかに一致すると、言葉ができたと判定される仕組みです。

言葉は縦（上から下）または横（左から右）に作ることができます。上手に文字を置くと、複数の言葉を同時に作ることもできます（Fig. 3-83）。例えば「あ」と「い」の間に「お」を置くと、

あお
あおい
おい

のように、3つの言葉を同時に作ることができます。縦と横に別の言葉を作ることもできるので、置き方によっては、かなり多くの言葉を一度に作ることが可能です。

言葉を作るゲームには『ことばのパズル もじぴったん』などがあります。このゲームでは、ステージに格子状に文字を並べて、言葉を作ります。ステージをクリアするには、決められた

個数以上の言葉を作る必要があります。

このゲームの特徴は、膨大な言葉の辞書を用意していることです。適当に文字を置いても、意外な言葉ができる面白さがあります。さらに、作った言葉の意味も表示されます。また、ステージごとに使える文字の種類が決まっていたり、あらかじめ置かれている文字の配置が変わっていたりと、プレイヤーを飽きさせないいろいろな工夫があります。

アルゴリズム

言葉を作るアクションを実現するポイントは、言葉ができたかどうかの判定方法です。

まず、ステージをセルで表現します（Fig. 3-84）。ここでは「あ」「い」「う」「え」「お」の文字を、「0～4」の数字で表しました。わかりやすくするために、図では「あ」～「お」の文字と「0～4」の数字の両方を示しています。

ボタンを押したら、カーソルの位置に新しい文字を置きます（Fig. 3-85）。すでに置かれている文字の上に重ねて置くことはできません。文字を置いたら、文字に対応する数字をセルに書き込みます。

Fig. 3-84　ステージをセルで表現する

「あ」～「お」の文字を0～4の数字で表す

			あ(0)	
	え(3)		い(1)	お(4)
い(1)	う(2)	い(1)		
あ(0)		う(2)	え(3)	
お(4)				

Fig. 3-85　新しい文字を書き込む

カーソルの位置に新しい文字を置く

			あ(0)	
	え(3)	あ(0)	い(1)	お(4)
い(1)	う(2)	い(1)		
あ(0)		う(2)	え(3)	
お(4)				

Fig. 3-86　言葉ができる可能性がある範囲

新しい文字

			あ(0)	
	え(3)	あ(0)	い(1)	お(4)
い(1)	う(2)	い(1)		
あ(0)		う(2)	え(3)	
お(4)				

言葉ができる可能性がある範囲（横方向）

言葉ができる可能性がある範囲（縦方向）

Fig. 3-87　文字列が辞書に載っているかどうかを調べる

			あ(0)	
	え(3)	あ(0)	い(1)	お(4)
い(1)	う(2)	い(1)		
あ(0)		う(2)	え(3)	
お(4)				

○「えあ」
×「えあい」
×「えあいお」
○「あい」
×「あいお」

文字列を順番に取り出し辞書に載っているかどうかを調べる（横方向）

○「あい」
×「あいう」

文字列を順番に取り出し辞書に載っているかどうかを調べる（縦方向）

言葉ができたかどうかの判定

新しい文字を置いたら、言葉ができたかどうかを判定します。縦方向と横方向について、順番に調べます。どちらを先に調べてもかまいません。

言葉の文字数は2文字から4文字までとしましょう。言葉ができる可能性があるのは、新しい文字を含んで、左右または上下に4文字までの範囲です（Fig. 3-86）。

この範囲から、2文字から4文字までの文字列を順番に取り出し、辞書に載っているかどうかを調べます（Fig. 3-87）。辞書に載っていたら、言葉ができたということです。

言葉ができたら、言葉の部分を特別な色で表示するなどして、プレイヤーに示すとよいでしょう。本書のサンプルでは、一定時間だけ言葉の部分を灰色で表示します。複数の言葉を作ったときには、見つかった順番に、一定時間ずつ灰色で表示します。

プログラム

List 3-10は言葉を作るプログラムです。ステージの移動処理を掲載しました。

移動処理は、初期状態・入力状態・探索状態・表示状態に分かれています。初期状態では、新しい文字をランダムに選んでから、入力状態に移行します。

入力状態では、レバー入力に応じて、カーソルを上下左右に動かします。ボタンを入力したときには、カーソルの位置に新しい文字を置きます。セルに新しい文字を書き込み、探索状態に移行します。

探索状態では、言葉ができたかどうかを判定します。新しい文字を中心として、2文字から4文字までの文字列を順番に取り出し、辞書の言葉との一致を調べます。一致したら、言葉の部分のセルにマークを付けて、表示状態に移行します。サンプルでは、コード内に辞書を直接埋め込んでいます。

表示状態では、一定時間が経過するのを待ちます。待っている間は、マークの付いたセルを特別な色で描画します。描画処理については、付録CD-ROMの「Puzzle￥Stage3.h」にあるCCrosswordStageクラスのDraw関数を参照してください。一定時間が経過したら、探索状態に戻ります。

探索状態では、縦方向と横方向に言葉を探します。すべての言葉を見つけたら、初期状態に移行します。

List 3-10 言葉を作る（CCrosswordStageクラス）

```
// ステージの移動処理
bool CCrosswordStage::Move(const CInputState* is) {

    // セルのサイズ
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();
```

```
//初期状態
if (State==0) {

    //新しい文字を選ぶ
    Kana=Rand.Int31()%CROSSWORD_KANA_COUNT;

    //入力状態に移行する
    State=1;
}

//入力状態
if (State==1) {

    //レバー入力に応じて、
    //カーソルを上下左右に動かす
    if (!PrevLever) {
        if (is->Left && CX>0) CX--; else
        if (is->Right && CX<xs-1) CX++; else
        if (is->Up && CY>0) CY--; else
        if (is->Down && CY<ys-1) CY++;
    }

    //前回のレバー入力を記録する
    PrevLever=is->Left||is->Right||is->Up||is->Down;

    //カーソルが文字のないセルにあるときに、
    //ボタンを押したら、
    //カーソルの位置に新しい文字を置く
    if (
        !PrevButton &&
        is->Button[0] &&
        Cell->Get(CX, CY)==
            CROSSWORD_KANA_COUNT
    ) {
        //新しい文字をセルに書き込む
        Cell->Set(CX, CY, Kana);

        //縦方向に言葉を探すための準備
        //探索の座標と方向、言葉の文字数を設定する
        X=CX;
        Y=max(0, CY-CROSSWORD_MAX_LENGTH+1);
        VX=0;
        VY=1;
        Length=CROSSWORD_MIN_LENGTH;

        //探索状態に移行する
        State=2;
    }
    PrevButton=is->Button[0];
```

```
}

//言葉の辞書
//（自由に言葉を追加してもかまわない）
static const wchar_t* dictionary[]={
    L"あい", L"あいあい", L"あう", L"あお", L"あおい",
    L"いあい", L"いい", L"いう", L"いえ", L"いえい", L"いお",
    L"うい", L"うえ", L"うお",
    L"えあ", L"えい",
    L"おあ", L"おい", L"おう", L"おおい", L"おおう",
    NULL
};

//探索状態
if (State==2) {

    //ステージから取得した言葉
    wchar_t word[CROSSWORD_MAX_LENGTH+1];

    //決められた範囲を探索する
    for (;
        X<=CX && Y<=CY;
        X+=VX, Y+=VY,
        Length=CROSSWORD_MIN_LENGTH
    ) {
        //ステージからはみ出さない最長の言葉の文字数を求める
        int max_length=min(
            CROSSWORD_MAX_LENGTH,
            (xs-X)*VX+(ys-Y)*VY);

        //最短から最長の文字数までを調べる
        for (; Length<=max_length; Length++) {

            //ステージのセルから文字を取り出して、
            //言葉の文字列を生成する
            int i, x, y;
            for (i=0, x=X, y=Y; i<Length; i++, x+=VX, y+=VY) {
                word[i]=L"あいうえお　"[Cell->Get(x, y)];
            }
            word[i]=L'¥0';

            //新しく置いた文字を含んでいるときの処理
            if (x>CX || y>CY) {

                //辞書の言葉との一致を調べる
                for (i=0; dictionary[i]; i++) {
                    if (wcscmp(word, dictionary[i])==0) break;
                }
```

```
                    // 辞書の言葉に一致したら、ループを抜け出す
                    if (dictionary[i]) break;
                }
            }

            // 辞書の言葉に一致したときには、このループも抜け出す
            if (Length<=max_length) break;
        }

        // 縦方向の探索が完了したとき
        if (Y>CY) {

            // 横方向に言葉を探すための準備
            // 探索の座標と方向、言葉の文字数を設定する
            X=max(0, CX-CROSSWORD_MAX_LENGTH+1);
            Y=CY;
            VX=1;
            VY=0;
            Length=CROSSWORD_MIN_LENGTH;
        } else

        // 横方向の探索が完了したとき
        if (X>CX) {

            // 初期状態に移行する
            State=0;
        } else

        // 辞書の言葉に一致したときの処理
        {
            // 一致した部分を異なる色で表示するために、
            // セルにマークを付ける
            for (int i=0, x=X, y=Y; i<Length; i++, x+=VX, y+=VY) {
                Cell->Set(x, y, Cell->Get(x, y)|0x80);
            }

            // より長い言葉を探す
            Length++;

            // タイマーを設定し、表示状態に移行する
            Time=0;
            State=3;
        }
    }

    // 表示状態
    if (State==3) {

        // タイマーの更新
```

```
        Time++;

        // 一定時間が経過したとき
        if (Time==30) {

            // 一致した部分の表示を終了するために、
            // セルのマークを外す
            for (int i=0, x=X, y=Y; i<Length; i++, x+=VX, y+=VY) {
                Cell->Set(x, y, Cell->Get(x, y)&0x7f);
            }

            // 探索状態に移行する
            State=2;
        }
    }

    return true;
}
```

SAMPLE

「CROSSWORD」は「言葉を作る」のサンプルです。レバーの上下左右（カーソルキーの上下左右）でカーソルが移動します。灰色で表示されている文字がカーソルです。

ボタン0（Zキー）を押すと、新しい文字をカーソルの位置に置くことができます。他の文字の上に置くことはできません。

新しい文字を置いたときに、その文字を使った言葉ができていると、言葉の部分が灰色で表示されます。言葉は縦または横に作ることができます。複数の言葉を同時に作ると、できた言葉が次々に表示されます。

CROSSWORD → p. 387

まとめ

本章では「つなぐ」アクションに注目して、線路やパイプ、あるいは言葉といったいろいろなものをつなぐゲームを紹介しました。さまざまなパズルゲームを見渡すと、実は「つなぐ」ことをテーマにしたゲームは数多く存在します。「つなぐ」というアクションは、与えられたパーツを使ってプレイヤーが何かを作る行為なので、ものを作り上げることに似た楽しさがあります。

というわけで、「つなぐアクションで、作る楽しみを味わわせよう！」というのが本章のまとめです。

Stage 04

ブロック

Block

「ブロック」は多くのパズルゲームで使われている要素です。同じようにブロックを使っていても、落としたり、並べたり、つなげたり、撃ったりと、ゲームによってまったく遊び方が違います。また、落としてぶつけたり、引き寄せて撃ったりといったように、複数の操作を組み合わせたゲームもあり、非常にバリエーションが豊かです。

ブロックを矩形にして消す

ブロックを撃ち出して結合し、矩形を作るアクションです。ブロックを矩形状に並べると、消すことができます。

ステージの上方から、いくつかのブロックが結合したものが降ってきます（Fig. 4-1）。結合の形には、カギ型、コの字型、T字型などがあります。

ステージの下方には、プレイヤーが操作できるキャラクターがいます。レバーの左右でキャラクターを左右に動かし、ボタンを押すとブロックを撃つことができます（Fig. 4-2）。

撃ち出したブロックは上に向かって飛んでいきます。他のブロックにぶつかると、ブロックは結合します（Fig. 4-3）。

Fig. 4-1 ブロックが降ってくる

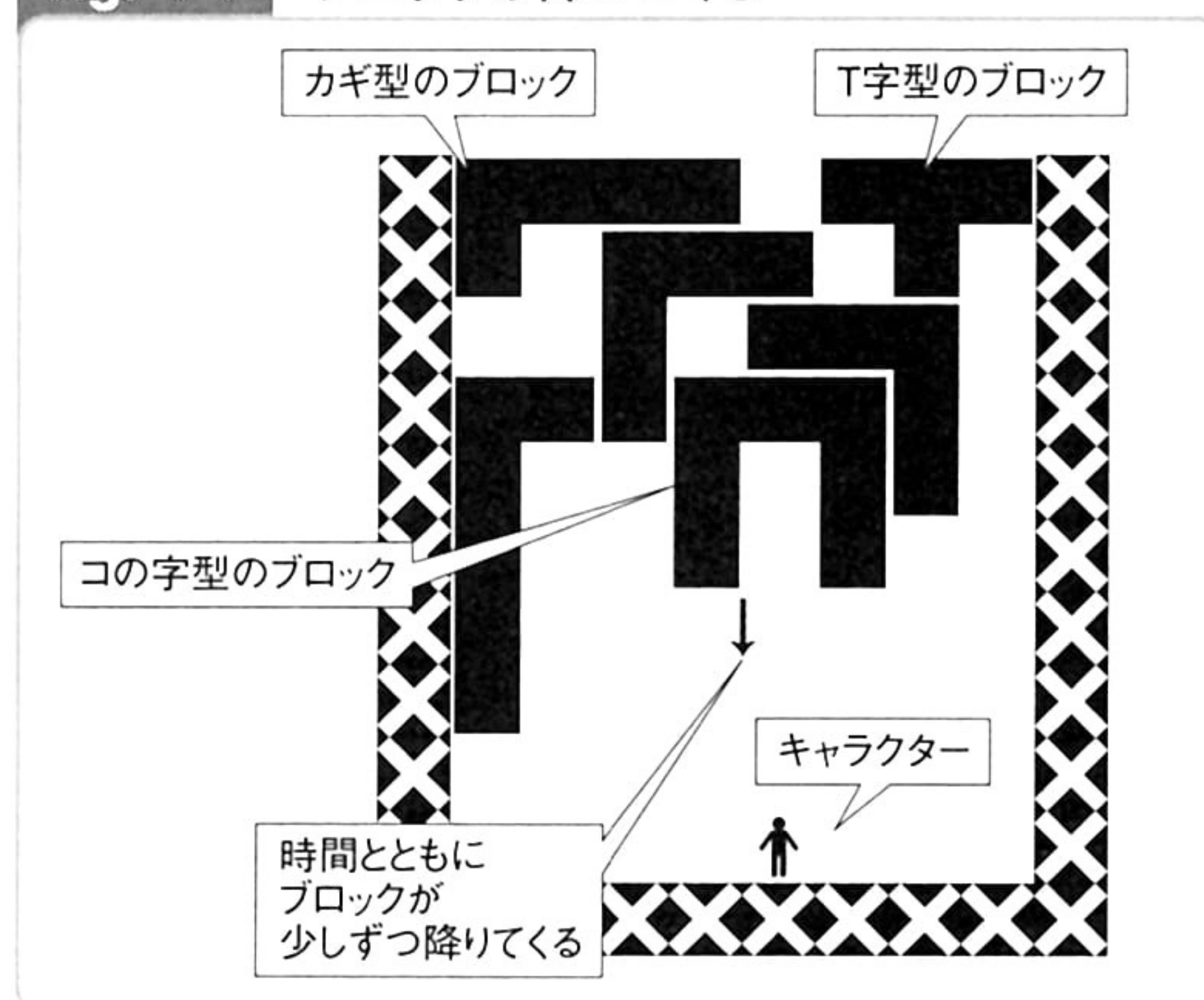



Fig. 4-2 ブロックを撃つ

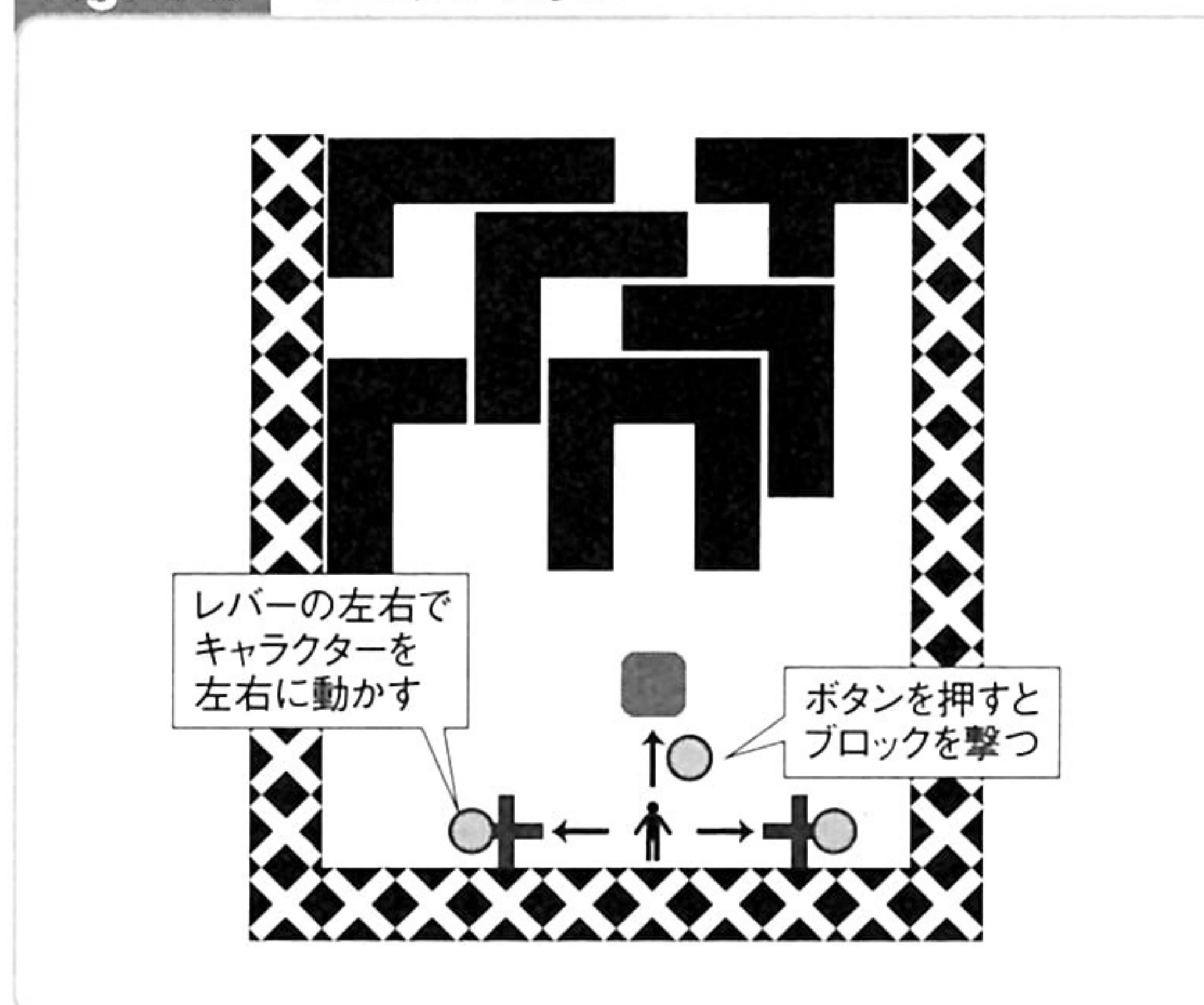



Fig. 4-3 ブロックを結合して矢印を作る

Fig. 4-4 ブロックが消える

次々にブロックを撃って結合させると、矩形を作ることができます。降ってくるブロックの形に応じて、欠けている部分を埋めるようにブロックを撃ち込んで、矩形を作ります。

矩形状に並んだブロックは消えます(Fig. 4-4)。降ってくるブロックがステージ下端に到達する前に、次々にブロックを撃ち込んで、素早く消していくことがゲームの目的です。

ブロックを矩形にして消すゲームには『クォース』があります。このゲームでは、降ってくるブロックに新しいブロックを撃ち込み、矩形を作って消します。ブロックがステージの下端に達するとミスになります。このゲームでは、本書のサンプルとは違い、結合していないブロックでも矩形状に並べると消せることがあります。

アルゴリズム

ブロックを矩形にして消すアクションを実現するには、ステージをセルで表現します。セルについては「迷路を歩く」(→p. 10)を参照してください。

ステージの壁を「=」で、新しいブロックを「+」で、落ちてくるブロックを数字で表すことにします(Fig. 4-5)。同じ数字のブロックは、結合していることを表します。

ブロックを描画するときには、上下左右のブロックを調べます(Fig. 4-6)。同じ種類のブロックが隣接している方向には、ブロックがつながるように描画します。

落ちてくるブロックは、時間とともに少しずつ落ちてきます。それには、一定時間ごとに、ステージ内のセルを1段ずつ下に移動させます(Fig. 4-7)。

プレイヤーがボタンを押したら、ステージ下端に新しいブロックを出現させます(Fig. 4-8)。そして、一定時間ごとに、ステージ内にある新しいブロックのセル(+)を、1段ずつ上に移動します(Fig. 4-9)。これで、新しいブロックは上に飛んでいきます。

新しいブロックを上に移動するときに、他のブロックにぶつかったら、新しいブロックをその位置に固定します(Fig. 4-10)。ブロックを結合させるために、ぶつかったブロックと同じ数字をセルに書き込みます。

Fig. 4-5　ステージをセルで表現する

=	0	0	0	0		1	1	1	=
=	0		2	2	2				
=			2		3	3	3		=
=	4	4	2	5	5	5	3		=
=	4			5		5	3		=
=	4			5		5			=
=	4								=
=	4								=
					+				
									=
=	=	=	=	=	=	=	=	=	=

壁(=)

落ちてくるブロック(数字)

新しいブロック(+)

同じ数字のブロックは結合している

Fig. 4-6　ブロックの描画

ブロックを描画するときには上下左右のブロックを調べる

同じ種類のブロックがある方向にはブロックがつながるように描画する

	3	3	3
5	5	5	3
5		5	3
5		5	

同じ種類　異なる種類　つなげる　つなげない

Fig. 4-7 ブロックを落とす

ブロックを落とすには一定時間ごとにセルを1段ずつ下に移動させる

Fig. 4-8 新しいブロックの出現

プレイヤーがボタンを押したら新しいブロックを出現させる

キャラクター

Fig. 4-9 新しいブロックを上に飛ばす

一定時間ごとに新しいブロックのセルを1段ずつ上に移動する

Fig. 4-10 新しいブロックを固定する

ぶつかったブロック

他のブロックにぶつかったら新しいブロックを固定する

新しいブロック

ブロックの種類はぶつかったブロックと同じにする

矩形状に並んでいるかどうかを確認する

次に、ブロックが矩形状に並んだかどうかを確かめます。新しく結合したブロックの位置から、上下左右に向かって、同じ種類のブロックが連続している範囲を調べます（Fig. 4-11）。

範囲を調べたら、矩形の範囲内に空のセルがあるかどうかを調べます（Fig. 4-12）。空のセルがある場合は、矩形が完成していないので、ブロックは消しません。

Fig. 4-11 ブロックが連続する範囲を調べる

同じ種類のブロックが連続する範囲を上下左右に向かって調べる

新しいブロック

Fig. 4-12 空のセルの有無を調べる

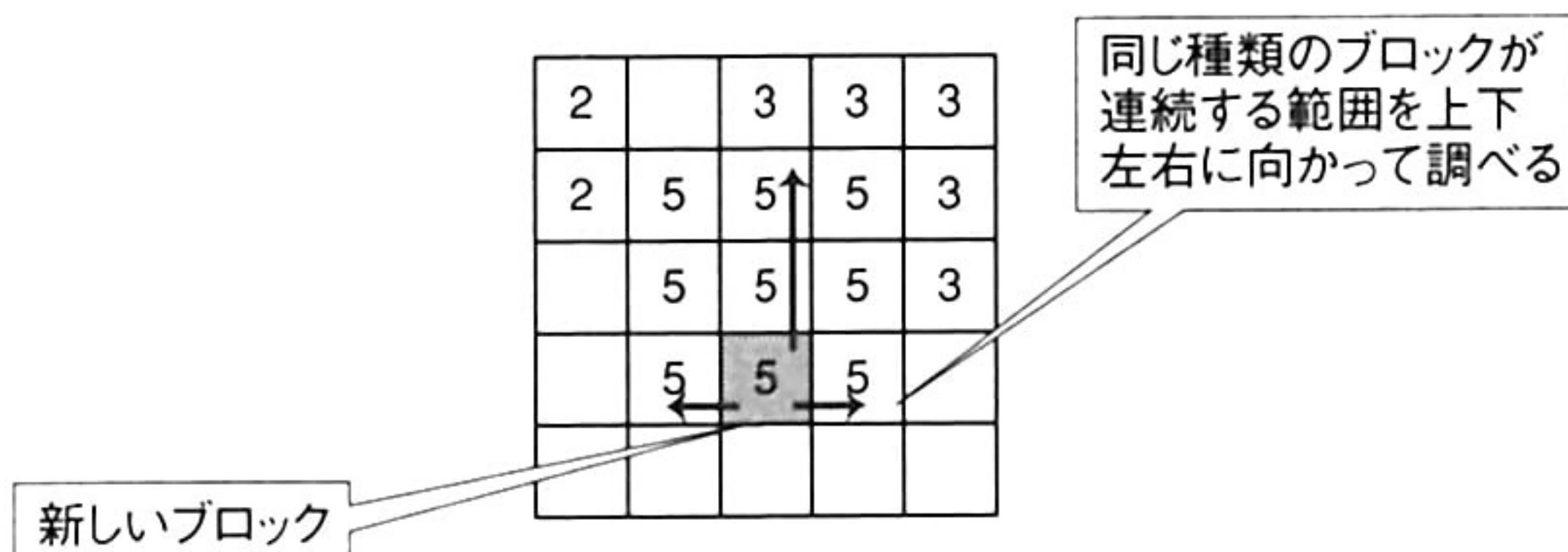
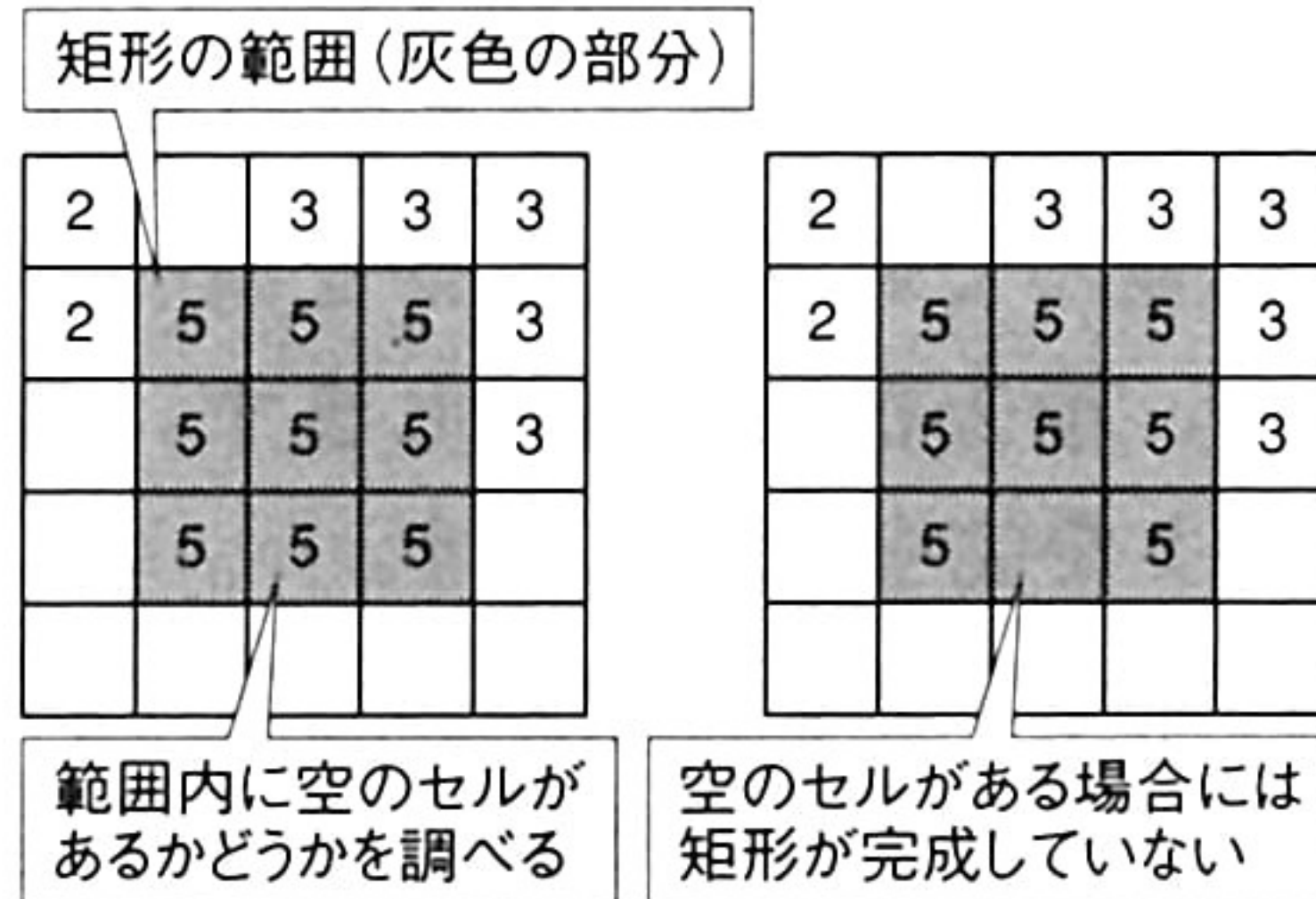
矩形の範囲（灰色の部分）

範囲内に空のセルがあるかどうかを調べる

空のセルがある場合には矩形が完成していない

Fig. 4-13 同じ種類のセルの有無を調べる

矩形の範囲（白色の部分）

範囲の1つ外側に範囲内と同じ種類のセルがあるかどうかを調べる

同じ種類のセルがある場合には矩形が完成していない

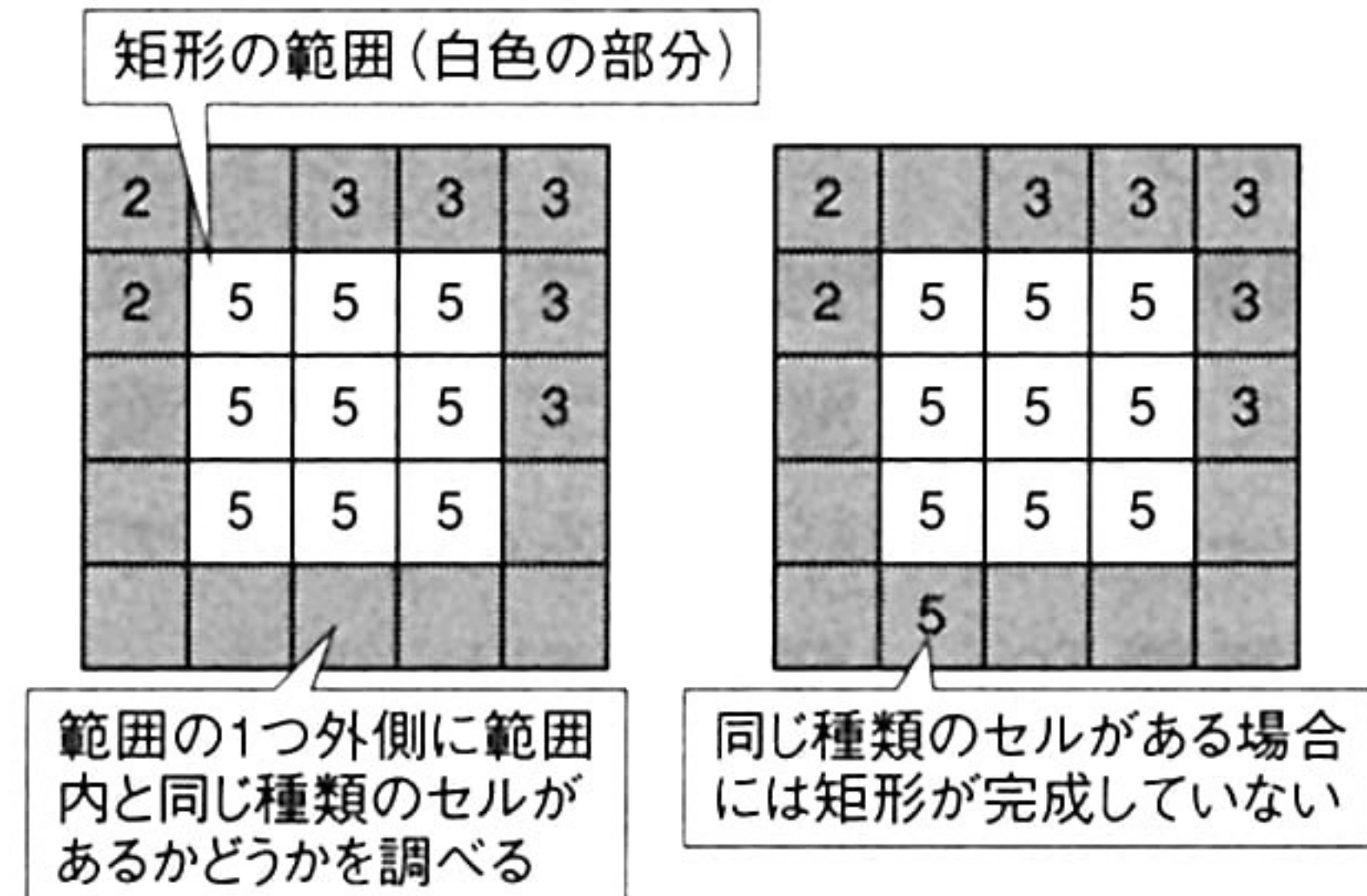



Fig. 4-14 ブロックを消えるブロックにする

同じ種類のブロックが矩形状に並んだら消えるブロック（*）にする

Fig. 4-15 消えるブロックを消す

一定時間が経過したら消えるブロックを空のセルにする

続いて、範囲の1つ外側に、範囲内と同じ種類のブロックのセルがあるかどうかを調べます（Fig. 4-13）。同じ種類のセルがある場合は、やはり矩形が完成していないので、ブロックは消しません。

空のセルも同じ種類のセルもない場合には、同じ種類のブロックが矩形状に並んだということなので、ブロックを消えるブロック（*）にします（Fig. 4-14）。消えるブロックは、一定時間が経過したら、空のセルにすることによって、完全に消します（Fig. 4-15）。完全に消えるまでの間は、ブロックがだんだん薄くなるような表示をすると、どのブロックが消えたのかがわかりやすくなります。

プログラム

List 4-1はブロックを矩形にして消すプログラムです。ステージの移動処理と描画処理、矩形状に並んだブロックを消す処理を掲載しました。

移動処理（Move関数）は、入力状態と消去状態に分かれています。入力状態では、レバー入力に応じてキャラクターを左右に動かします。ボタン入力があったら、新しいブロックを出現

させます。また、新しいブロックを上に移動する処理と、落ちてくるブロックを一定時間ごとに落下させる処理も行います。

新しいブロックが他のブロックにぶつかったら、矩形状に並んだブロックを消す処理（EraseBlock関数）を呼び出します。ここではブロックが矩形状に並んでいるかどうかを調べて、矩形ができていたら、消えるブロックに変化させます。そして、消去状態に移行します。消去状態では、一定時間が経過するのを待ってから、消えるブロックを完全に消します。

描画処理（Draw関数）では、ステージの壁、ブロック、キャラクターを描画します。落ちてくるブロックについては、同じ種類のブロックが隣接しているかどうかを調べて、隣接している方向にはつながるように描画しています。

本章のサンプルでは、プログラムを簡潔にするために、キャラクターやブロックの動きをセル単位にしています。より滑らかな動きにするには、「ブロックを落とす」（→p. 50）などを参考に、タイマーを使って描画座標を少しずつ変化させるとよいでしょう。

List 4-1 ブロックを矩形にして消す（CRectangleShapedBlockクラス）

```
// ステージの移動処理
bool CRectangleShapedBlock::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 入力状態
    if (State==0) {

        // レバー入力に応じてキャラクターを左右に動かす
        if (!PrevLever) {
            if (is->Left && CX>0) CX--;
            if (is->Right && CX<xs-1) CX++;
        }
        PrevLever=is->Left||is->Right;

        // ボタンを押したらブロックを撃つ
        if (!PrevButton && is->Button[0] && Cell->Get(CX, CY-1)==' ') {

            // キャラクターの位置に新しいブロックを出現させる
            Cell->Set(CX, CY, '+');
        }
        PrevButton=is->Button[0];

        // ステージ上端のセルを空にする
        for (int x=1; x<xs-1; x++) {
            Cell->Set(x, 0, ' ');
        }

        // 新しいブロックのセル（＋）を上に移動させる
        for (int y=1; y<ys-1; y++) {
```

```
    for (int x=1; x<xs-1; x++) {

        // 新しいブロックのセルが見つかったときの処理
        if (Cell->Get(x, y)=='+') {
            char c=Cell->Get(x, y-1);

            // 新しいブロックの上のセルが空ならば、
            // 新しいブロックを1つ上に移動する
            if (c==' ') {
                Cell->Set(x, y-1, '+');
                Cell->Set(x, y, ' ');
            } else

            // 新しいブロックの上のセルが空でなければ、
            // 新しいブロックを現在の位置に固定する
            {
                Cell->Set(x, y, c);

                // ブロックを消す処理を呼び出す
                EraseBlock(x, y);
            }
        }
    }
}

// 一定時間ごとにブロックを落下させる
DropTime++;
if (DropTime==60) {
    DropTime=0;
    int x, y;

    // ステージの下端が空でなければ、
    // ブロックを落下させない
    for (x=1; x<xs-1; x++) {
        if (Cell->Get(x, ys-3)!=' ') break;
    }

    // ステージの下端が空ならば、
    // ブロックを落下させる
    if (x==xs-1) {

        // すべてのセルを1段ずつ下に移動する
        for (y=ys-3; y>0; y--) {
            for (x=1; x<xs-1; x++) {
                Cell->Set(x, y, Cell->Get(x, y-1));
            }
        }
    }
}
```

```
	}

	// 消去状態の処理
	if (State==1) {

		// 一定時間が経過したら、ブロックを消す
		Time++;
		if (Time==30) {

			// 矩形に並んだブロックを消す
			for (int y=T; y<B; y++) {
				for (int x=L; x<R; x++) {
					Cell->Set(x, y, ' ');
				}
			}

			// 入力状態に移行する
			State=0;
		}
	}

	return true;
}

// 矩形状に並んだブロックを消す処理
void CRectangleShapedBlock::EraseBlock(int x, int y) {

	// セルの個数
	int xs=Cell->GetXSize(), ys=Cell->GetYSize();

	// 現在位置のセルを取得する
	char c=Cell->Get(x, y);

	// 現在位置から左・右・上・下へ、
	// ブロックが連続している範囲を調べる
	for (L=x; Cell->Get(L-1, y)==c; L--) ;
	for (R=x+1; Cell->Get(R, y)==c; R++) ;
	for (T=y; Cell->Get(x, T-1)==c; T--) ;
	for (B=y+1; Cell->Get(x, B)==c; B++) ;

	// 矩形の範囲内に、空のセルがあるかどうかを調べる
	for (y=T; y<B; y++) {
		for (x=L; x<R; x++) {
			if (Cell->Get(x, y)!=c) break;
		}
		if (x<R) break;
	}

	// 空のセルがない場合の処理
```

```
    if (y==B) {

        // 矩形の範囲の1つ外側に、
        // ブロックのセルがあるかどうかを調べる
        for (x=L; x<R; x++) {
            if (Cell->Get(x, T-1)==c || Cell->Get(x, B)==c) break;
        }
        for (y=T; y<B; y++) {
            if (Cell->Get(L-1, y)==c || Cell->Get(R, y)==c) break;
        }

        // ブロックのセルがない場合の処理
        if (x==R && y==B) {

            // 矩形の範囲を消えるブロック(*)にする
            for (int y=T; y<B; y++) {
                for (int x=L; x<R; x++) {
                    Cell->Set(x, y, '*');
                }
            }

            // タイマーを設定し、消去状態に移行する
            Time=0;
            State=1;
        }
    }
}

// ステージの描画処理
void CRectangleShapedBlock::Draw() {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 描画サイズ
    float
        sw=Game->GetGraphics()->GetWidth()/xs,
        sh=Game->GetGraphics()->GetHeight()/ys;

    // 壁のセル(=)を描画する
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {
            if (Cell->Get(x, y)=='=') {
                Game->Texture[TEX_FLOOR]->Draw(
                    x*sw, y*sh, sw, sh, 0, 0, 1, 1, COL_BLACK);
            }
        }
    }
```

```
//ブロックのセルを描画する
for (int y=1; y<ys-1; y++) {
    for (int x=1; x<xs-1; x++) {
        char c=Cell->Get(x, y);

        //新しいブロック(+)の描画
        if (c=='+') {
            Game->Texture[TEX_OBJECT]->Draw(
                x*sw, y*sh, sw, sh, 0, 0, 1, 1, COL_LGRAY);
        } else

        //その他のブロックの描画
        if (c!=' ') {

            //上下左右に同じ種類のブロックが
            //隣接しているかどうかによって描画座標を調整し、
            //ブロックの結合を表現する
            static const float gap=0.1f;
            float l=x, r=x+1, t=y, b=y+1;
            if (c!=Cell->Get(x-1, y)) l+=gap;
            if (c!=Cell->Get(x+1, y)) r-=gap;
            if (c!=Cell->Get(x, y-1)) t+=gap;
            if (c!=Cell->Get(x, y+1)) b-=gap;

            //消えるブロックは時間とともに薄くなるように描き、
            //通常のブロックは黒色で描く
            float f=(float)Time/30;
            D3DCOLOR color=COL_BLACK;
            if (c=='*') color=D3DXCOLOR(f, f, f, 1);
            Game->Texture[TEX_FILL]->Draw(
                l*sw, t*sh, (r-l)*sw, (b-t)*sh, 0, 0, 1, 1, color);
        }
    }
}

//キャラクターの描画
Game->Texture[TEX_MAN]->Draw(
    CX*sw, CY*sh, sw, sh, 0, 0, 1, 1, COL_BLACK);
}
```

SAMPLE

「RECTANGLE SHAPED BLOCK」は「ブロックを矩形にして消す」のサンプルです。

レバーの左右(カーソルキーの左右)でキャラクターが移動します。ボタン0(Zキー)を押すと、新しいブロックを撃つことができます。

ステージ上方からは、ブロックが時間とともに少しずつ落ちてきます。ステージ下端にブロックが達してもミスにはなりませんが、ブロックの落下は止まります。

新しいブロックを落ちてくるブロックに当てると、ブロックは結合します。ブロックを上手に結合させて、矩形状に並べると、消すことができます。

RECTANGLE SHAPED BLOCK → p. 388

ブロックを変形させる

ボタン操作でブロックの形を変えるアクションです。「ブロックを落とす」(→p. 50) に似ていますが、ブロックの形を自由に変えられることが違います。

ステージ上方にはブロックが積まれています (Fig. 4-16)。本書のサンプルでは積まれたブロックは動きませんが、時間とともにブロックが落ちてくるようにしてもよいでしょう。

ステージ下方には新しいブロックが出現します (Fig. 4-17)。このブロックは時間とともに上昇します。レバーを上に入れると、上昇が速くなります。また、レバー入力で左右に動かすことができます。

ボタンを押すと、新しいブロックの一部が上に伸びます (Fig. 4-18)。ボタンを押すたびに、1段ずつブロックが伸びていきます。押すボタンによって、伸びる左右の位置が変わります。本書のサンプルでは、3つのボタン (ボタン0〜2) を使って、それぞれ左・中央・右が伸びるようにしました。

新しいブロックが他のブロックに接触すると、その場に固定されます。ブロックが横方向に1段揃うと、その段が消えます (Fig. 4-19)。ブロックの落下位置を調整するのに加えて、適切にブロックを変形させれば、効率よくブロックを消すことができます。

ブロックを変形させるアクションを採用したゲームには『アクアラッシュ』があります。このゲームでは、ステージ上方からブロックが落ちてきます。ブロックが下まで落ちる前に、下から新しいブロックをはめ込んで、次々にブロックを消すことがゲームの目的です。ブロックはボタン操作で変形させることができます。

Fig. 4-16 積まれたブロック

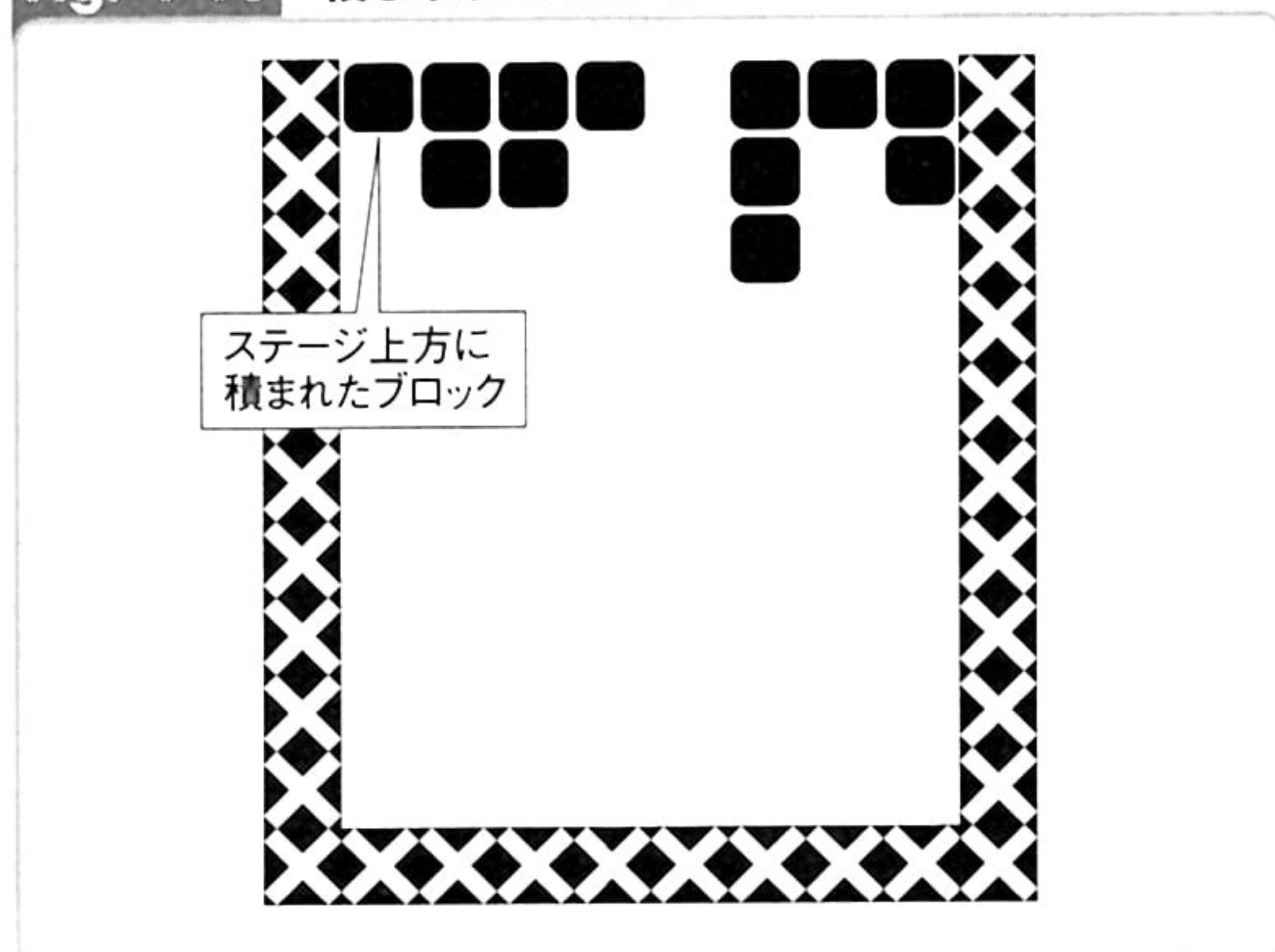



Fig. 4-17 新しいブロック

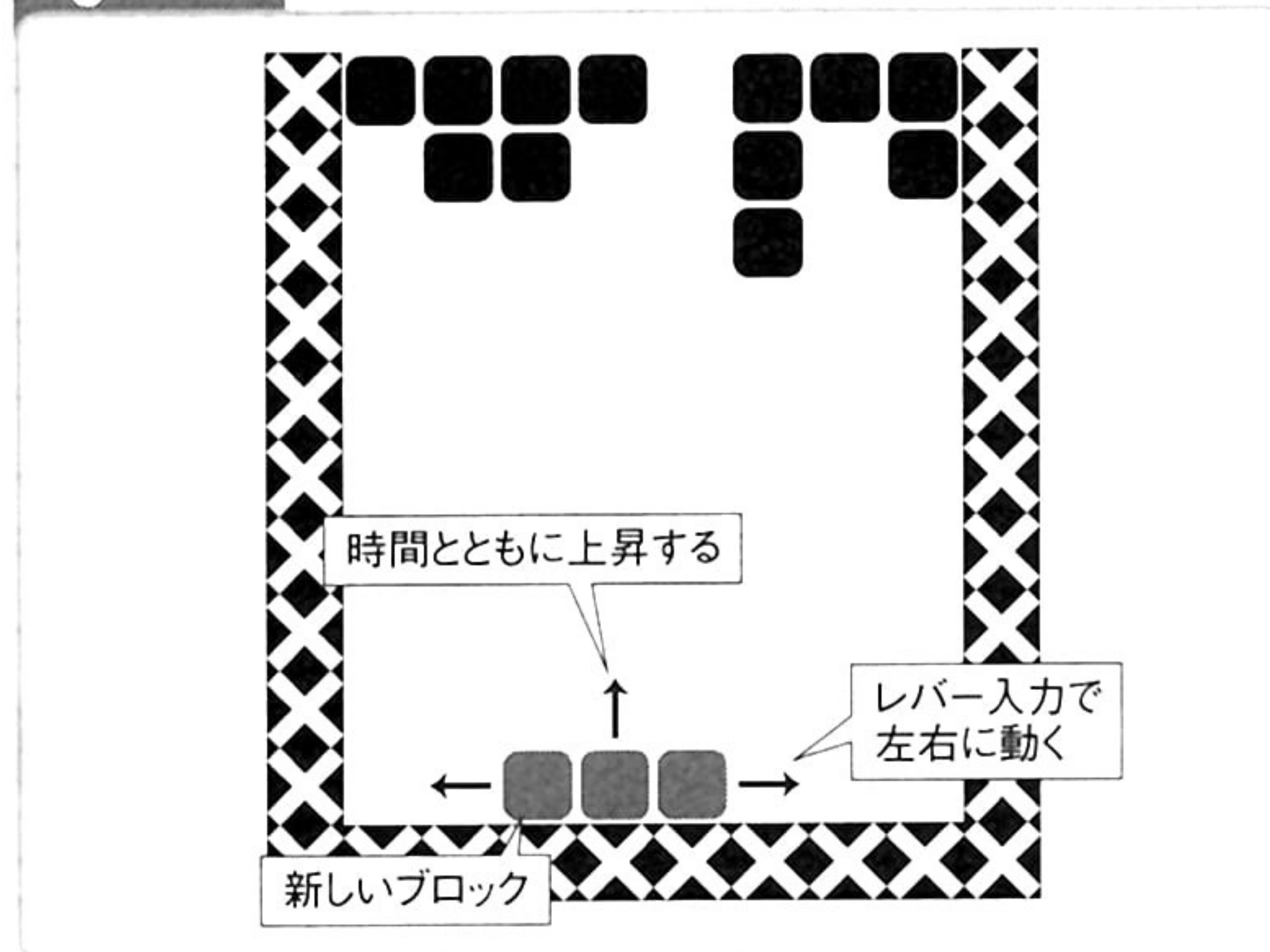



Fig. 4-18 ブロックが伸びる

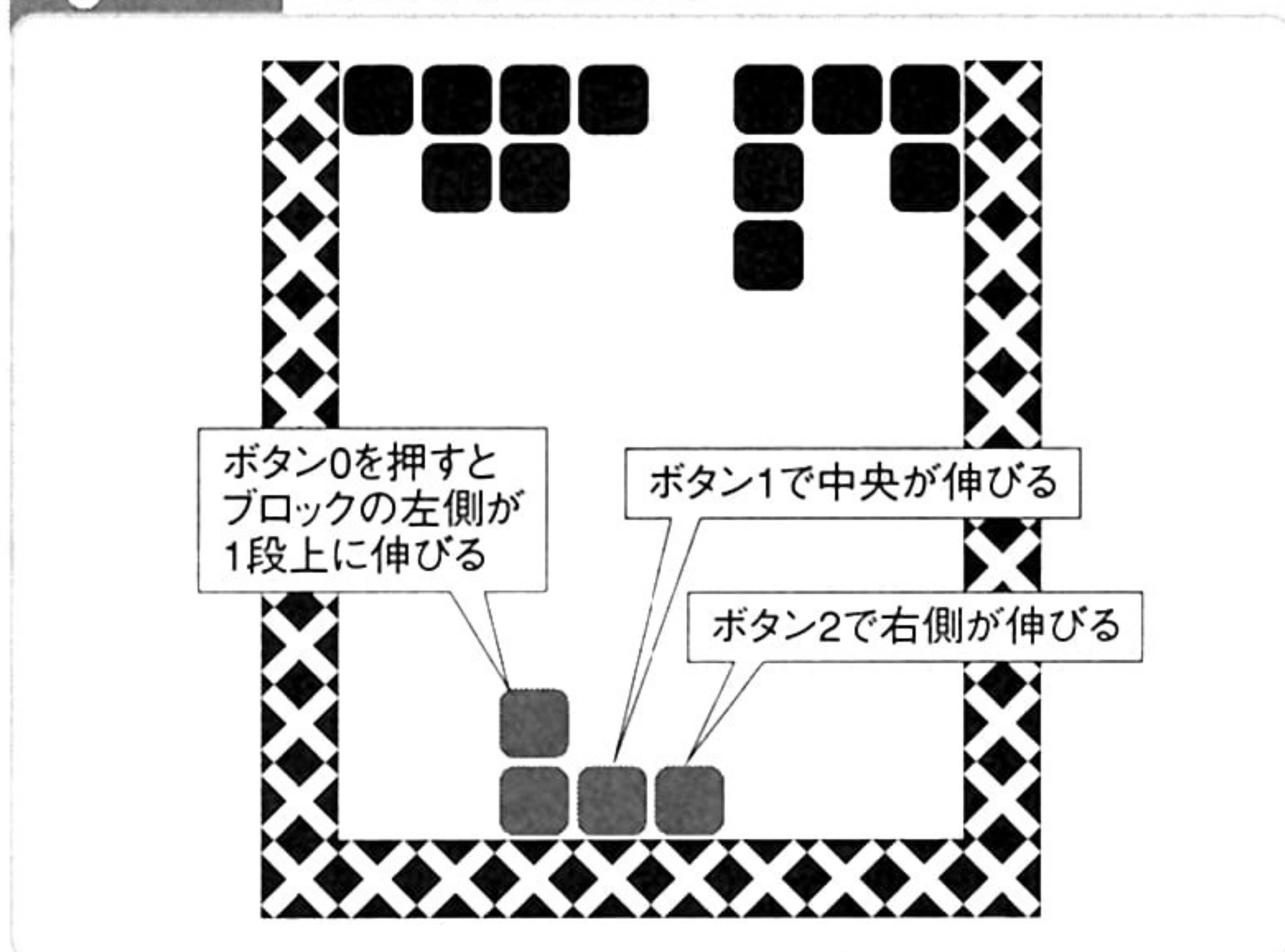
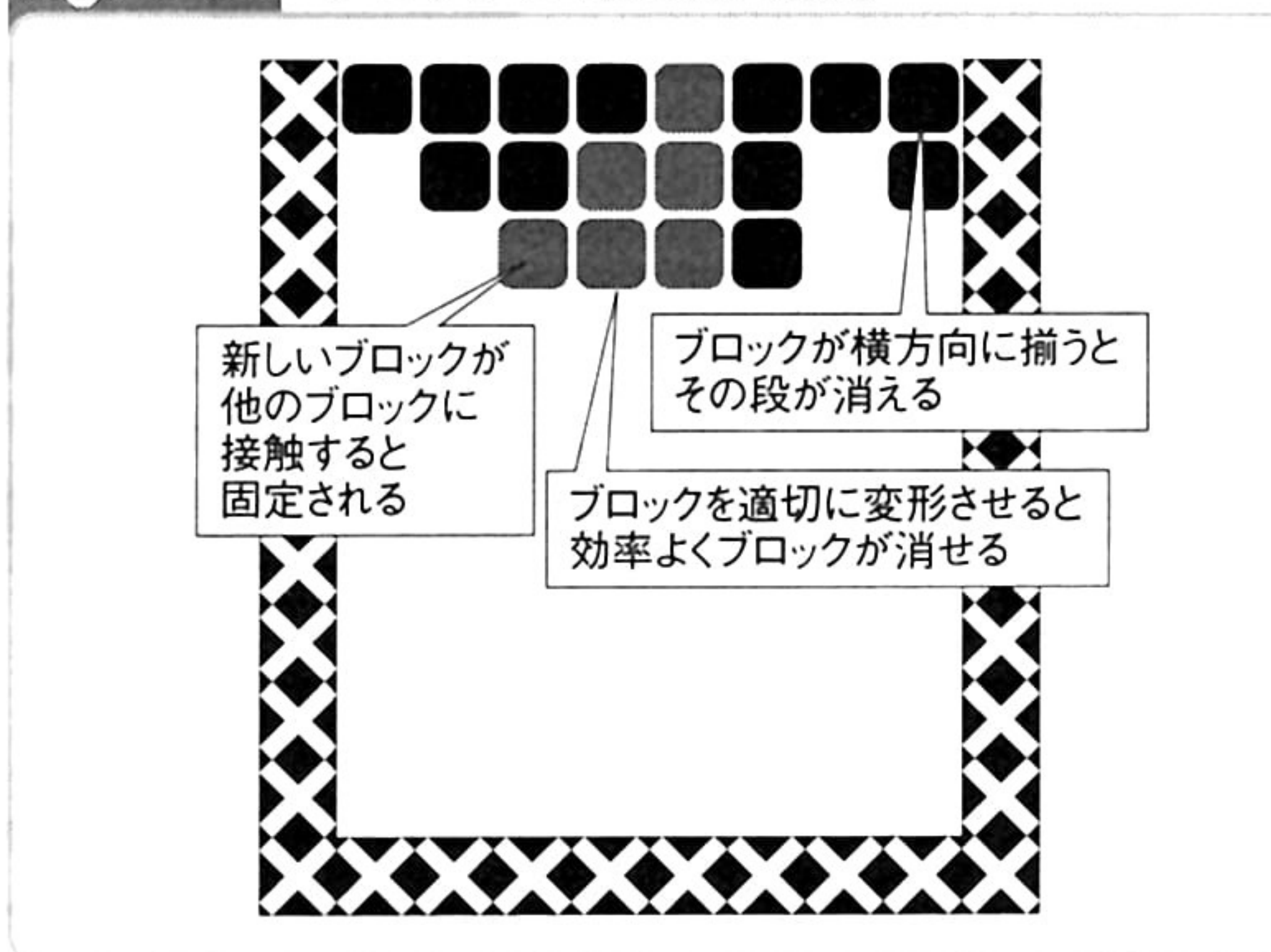



Fig. 4-19 ブロックを1段揃えて消す

アルゴリズム

ブロックを変形させるには、ブロックとステージをセルで表現します(Fig. 4-20)。ここではステージの壁を「=」で、ブロックを「#」で表すことにします。灰色で示した「#」の部分は新しいブロックです。

ボタンを押したときには、新しいブロックを変形させます(Fig. 4-21)。例えば、ボタン0を押した場合には、ブロックの左側を1段上に伸ばします。ボタン1を押したら中央を、ボタン2を押したら右側を伸ばすことにします。

新しいブロックは時間とともに上昇します。他のブロックに接触したら、その位置に新しいブロックを固定します(Fig. 4-22)。そして、横方向にブロックが1段揃ったかどうかを調べ、もしも揃っていたら、その段を消します。これは「ブロックを1段揃えて消す」(→p. 64)と同じ要領です。

Fig. 4-20 ブロックとステージをセルで表現する

=	#	#	#	#		#	#	#	=
=		#	#			#		#	=
=						#			=
=									=
=									=
=									=
=									=
=									=
=									=
=			#	#	#				=
=	=	=	=	=	=	=	=	=	=

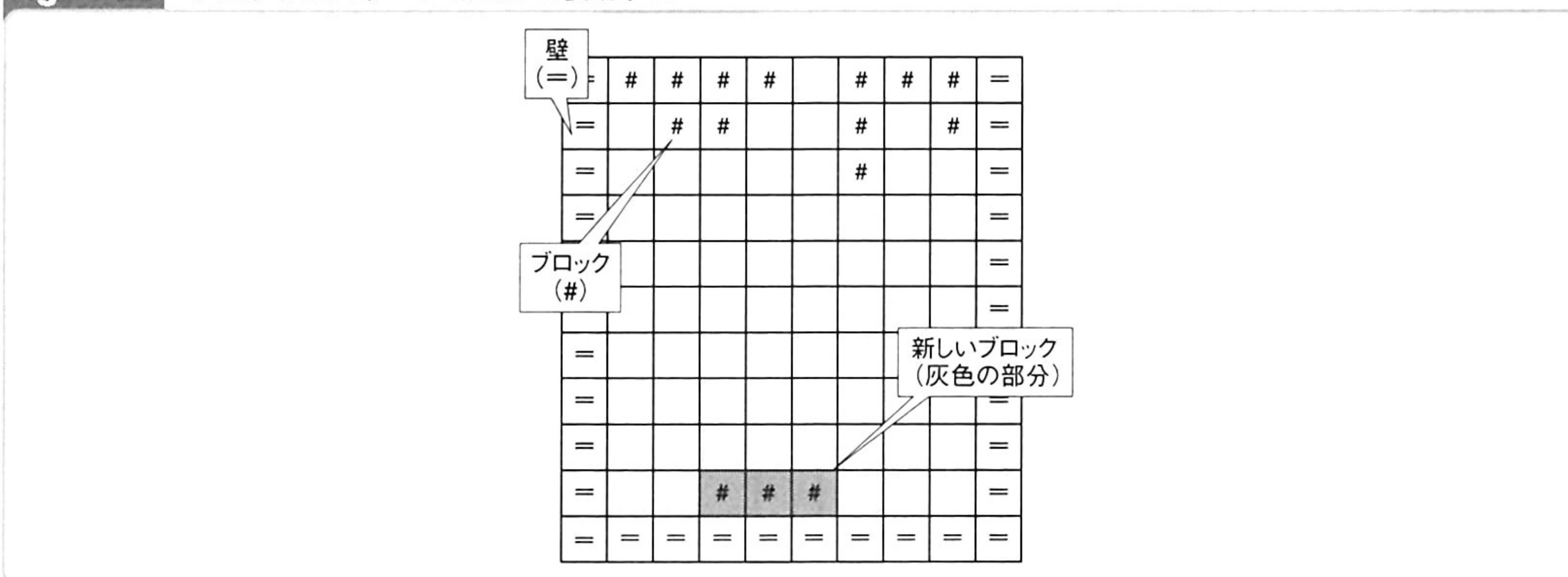
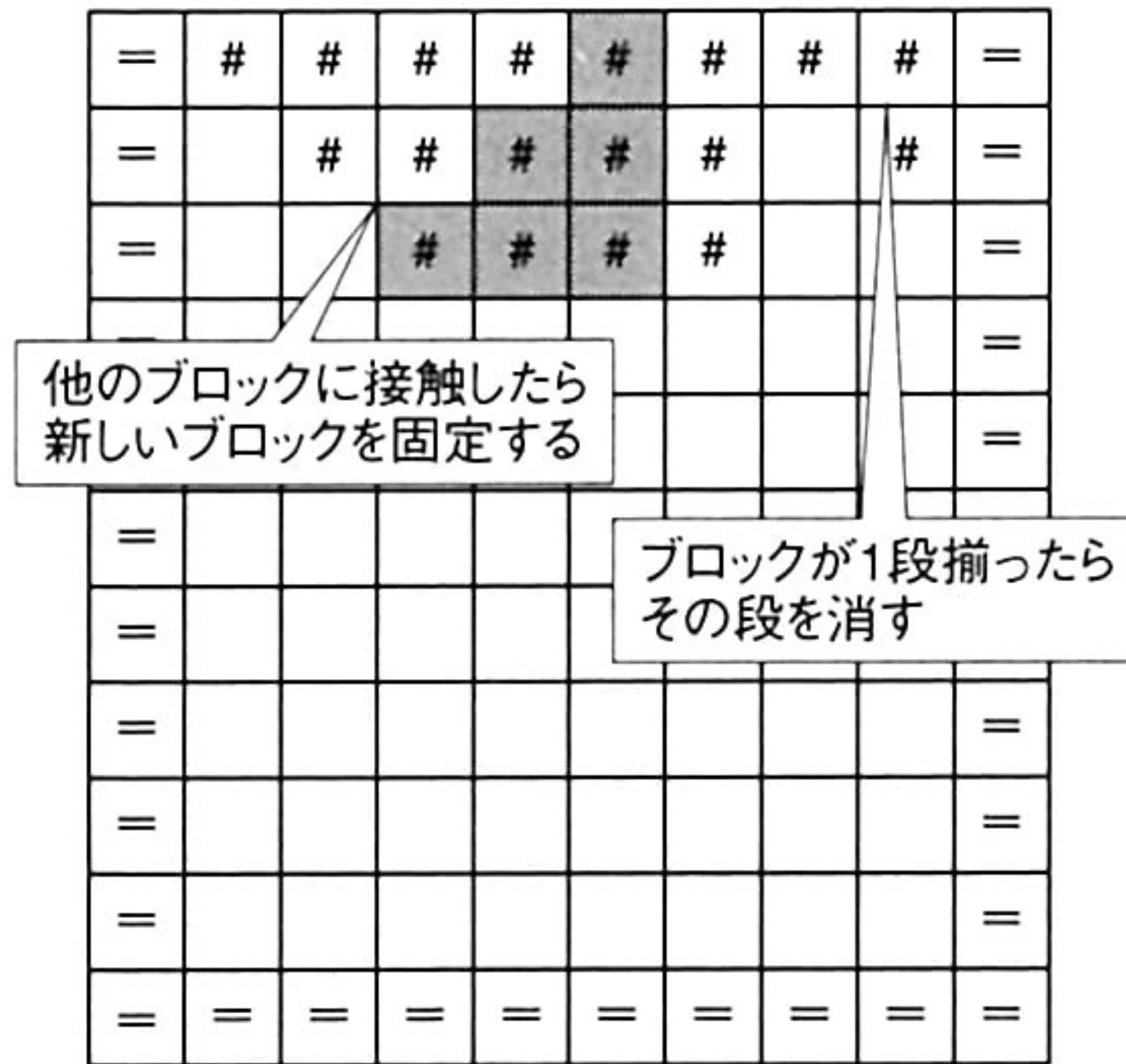



Fig. 4-21 ブロックの変形

=	#	#	#	#		#	#	#	=
=		#	#			#		#	=
=						#			=
=									=
=									=
=									=
=									=
=									=
=			#						=
=			#	#	#				=
=	=	=	=	=	=	=	=	=	=

Fig. 4-22 ブロックの固定

=	#	#	#	#	#	#	#	#	=
=		#	#	#	#	#		#	=
=			#	#	#	#			=
									=
									=
=									
=									
=									=
=									=
=									=
=	=	=	=	=	=	=	=	=	=

プログラム

List 4-2はブロックを変形させるプログラムです。ステージの移動処理を掲載しました。

このプログラムでは、新しいブロックのセル（BlockCell）と、ステージのセル（StageCell）を別々に用意しています。別々にした方が、ブロックの移動や当たり判定処理が簡単だったためです。別々にせずに、同じセルで扱うこともできます。この場合は、新しいブロックと他のブロックを区別するために、新しいブロックのセルを表す文字（#）を、他のブロックとは異なる文字にするとよいでしょう。

移動処理は、初期状態・入力状態・消去状態に分かれています。初期状態では、新しいブロックを初期化し、ステージの下端に出現させます。新しいブロックが他のブロックに接触する場合には、ブロックを出現させません。一般のゲームでは、これは他のブロックがステージ下端まで落ちてきてしまった状態なので、ミスになります。

入力状態では、レバー入力に応じて新しいブロックを左右に動かすとともに、一定時間ごとに上昇させます。新しいブロックが他のブロックに接触した場合には、ブロックを固定して、横方向にブロックが揃った段を探します。ブロックが揃ったら、その段のブロックを消えるブロックに変化させ、消去状態に移行します。

ボタンを押したときには、新しいブロックを変形させます。ボタン0〜2で、それぞれブロックの左・中央・右を1段ずつ上に伸ばします。他のブロックに接触する場合には、ブロックを伸ばしません。

消去状態では、一定時間が経過するのを待ってから、消えるブロックを完全に消します。時間を待つのは、この間に消えるブロックがだんだん薄くなって消えていくような演出をするためです。詳細はステージの描画処理（CTransformedBlockStageクラスのDraw関数）を参照してください。ブロックを消したら、初期状態に移行して、新しいブロックを出現させます。

List 4-2 ブロックを変形させる(CTransformedBlockStageクラス)

```
// ステージの移動処理
bool CTransformedBlockStage::Move(const CInputState* is) {

    // ステージのセルの個数、ブロックのセルの個数
    int
        xs=StageCell->GetXSize(), ys=StageCell->GetYSize(),
        bxs=BlockCell->GetXSize(), bys=BlockCell->GetYSize();

    // 初期状態
    if (State==0) {

        // ブロックのセルを空にする
        BlockCell->Clear(' ');

        // ブロックのセルの最下段に、横1列にブロックを書き込む
        for (int x=0; x<bxs; x++) {
            BlockCell->Set(x, bys-1, '#');
        }

        // ブロックのセルの初期位置を設定する
        CX=(xs-bxs)/2;
        CY=-1;

        // 新しいブロックが他のブロックに
        // 接触するかどうかを調べる
        if (!StageCell->Hit(CX, CY, BlockCell)) {

            // 接触しない場合には、入力状態に移行する
            DropTime=0;
            PrevUp=true;
            State=1;
        }
    }

    // 入力状態
    if (State==1) {

        // レバー入力に応じて、新しいブロックを左右に動かす
        if (!PrevLever) {
            if (is->Left &&
                !StageCell->Hit(CX-1, CY, BlockCell)) CX--;
            if (is->Right &&
                !StageCell->Hit(CX+1, CY, BlockCell)) CX++;
        }
        PrevLever=is->Left|is->Right;

        // 一定時間ごとに、新しいブロックを上昇させる
        // レバーを上に入れたときには、時間待ちをせずに上昇させる
```

```
DropTime++;
if (DropTime==60 || (!PrevUp && is->Up)) {
    DropTime=0;

    // 新しいブロックが他のブロックに接触しない場合は、
    // ブロックを上昇させる
    if (!StageCell->Hit(CX, CY-1, BlockCell)) {
        CY--;
    } else

    // 新しいブロックが他のブロックに接触する場合
    {
        // ブロックをステージに固定する
        StageCell->Merge(CX, CY, BlockCell);

        // ブロックが消えない場合には、
        // 初期状態に移行する
        State=0;

        // ブロックが消えるかどうかを調べる
        for (int y=2; y<ys-1; y++) {

            // 横方向にブロックが
            // 1段揃っているかどうかを調べる
            int x;
            for (x=1; x<xs-1; x++) {
                if (StageCell->Get(x, y)!='#') break;
            }

            // 1段揃っている場合の処理
            if (x==xs-1) {

                // ブロックを消えるブロックに変える
                for (x=1; x<xs-1; x++) {
                    StageCell->Set(x, y, '+');
                }

                // 消去状態に移行する
                Time=0;
                State=2;
            }
        }
    }
}

// レバーを入れっぱなしにしたときに、
// ブロックが連続して移動しないようにするための処理
if (!is->Up) PrevUp=false;

// ボタンを押したときには、ブロックを変形させる
```

```
	if (!PrevButton) {

		// どのボタンが押されたのかを調べる
		int x;
		for (x=0; !is->Button[x] && x<bxs; x++) ;

		// 有効なボタンが押されたときの処理
		if (x<bxs) {

			// ステージのセルに接触しなければ、
			// ボタンに対応する列のブロックを1段上に伸ばす
			for (int y=0; y<bys; y++) {
				if (
					BlockCell->Get(x, y)=='#' &&
					StageCell->Get(CX+x, CY+y-1)==' '
				) {
					BlockCell->Set(x, y-1, '#');
				}
			}
		}
	}

	// ボタンを押しっぱなしにしたときに、
	// ブロックが連続して変形しないようにするための処理
	PrevButton=false;
	for (int i=0; i<bxs; i++) {
		PrevButton=PrevButton||is->Button[i];
	}
}

// 消去状態
if (State==2) {

	// 一定時間が経過したら、消えるブロックを完全に消す
	Time++;
	if (Time==30) {

		// 消えるブロック（十）を探す
		for (int y=ys-3; y>=2; y--) {

			// 消えるブロックが見つかったら、
			// その段よりも下にある段を、
			// 1段ずつ上に移動する
			if (StageCell->Get(1, y)=='+') {
				for (int x=1; x<xs-1; x++) {
					for (int i=y; i<ys-2; i++) {
						StageCell->Set(
							x, i, StageCell->Get(x, i+1));
					}
```

```
                        // 最下段は空にする
                        StageCell->Set(x, ys-2, ' ');
                    }
                }
            }

            // 初期状態に移行する
            State=0;
        }
    }

    return true;
}
```

SAMPLE

「TRANSFORMED BLOCK」は「ブロックを変形させる」のサンプルです。

レバーの左右（カーソルキーの左右）でブロックが移動します。ブロックは時間とともに、自動的に上昇します。レバーの上（カーソルキーの上）を入力すると、速く上昇します。また、ボタン0～2（Zキー、Xキー、Cキー）を押すと、それぞれブロックの左・中央・右を1段ずつ上に伸ばすことができます。

ブロックが他のブロックに接触すると、その場所に固定されます。ブロックが横方向に1段揃うと、揃った段が消えます。積まれたブロックの形に合わせて、上手にブロックを変形させると、効率よくブロックを消すことができます。

TRANSFORMED BLOCK → p. 388

ブロックをぶつけて壊す

ブロックを積まれたブロックの上に落として、壊すアクションです。ぶつかったブロックにはひびが入ります。ひびが入ったブロックに、もう一度ブロックをぶつけると、ブロックを壊して消すことができます。

ステージにはブロックが積まれています（Fig. 4-23）。ステージ上方には新しいブロックが出現します。新しいブロックは、時間とともに少しずつ落下します。レバーを下に入力すると、落下スピードが上がります。また、レバー入力で左右に動かすこともできます。

新しいブロックを同じ種類のブロックの上に落とすと、ブロックを壊すことができます（Fig. 4-24）。ひびが入っていないブロックには、ひびが入ります。ひびが入っているブロックは、壊れて消えます。

ブロックをぶつけた衝撃は、隣接する同じ種類のブロックにも広がります（Fig. 4-25）。例えば、黒いブロック同士をぶつけると、縦・横・斜めに隣接する他の黒いブロックも、ひびが入

ったり壊れたりします。同じ種類のブロックが隣接しているかぎり、どこまでも衝撃が伝わります。

ブロックをぶつけて壊すゲームには『エメラルディア』があります。このゲームでは、さまざまな色のブロックが3個1組で降ってきます。同じ色のブロック同士をぶつけると、ブロックにひびを入れたり、壊したりすることができます。ブロックを次々に壊して消し、ステージがブロックで埋まらないようにすることが、ゲームの目的です。

本書のサンプルでは、プログラムを簡潔にするために、新しいブロックを1個ずつ降らせています。「ボールを落とす」(→p. 90) の手法を応用して、複数のブロックを同時に降らせることもできます。実際のゲームでは、複数のブロックを1組にして降らせるとよいでしょう。

また『エメラルディア』では、ブロックが消えたときに、落ちてきたブロックが同じ種類のブロックにぶつかると、連鎖的にブロックが壊れます。本書のサンプルには採用していませんが、「連鎖的に消す」(→p. 106) を応用して、連鎖的にブロックを壊すこともできます。

Fig. 4-23 積まれたブロックと新しいブロック

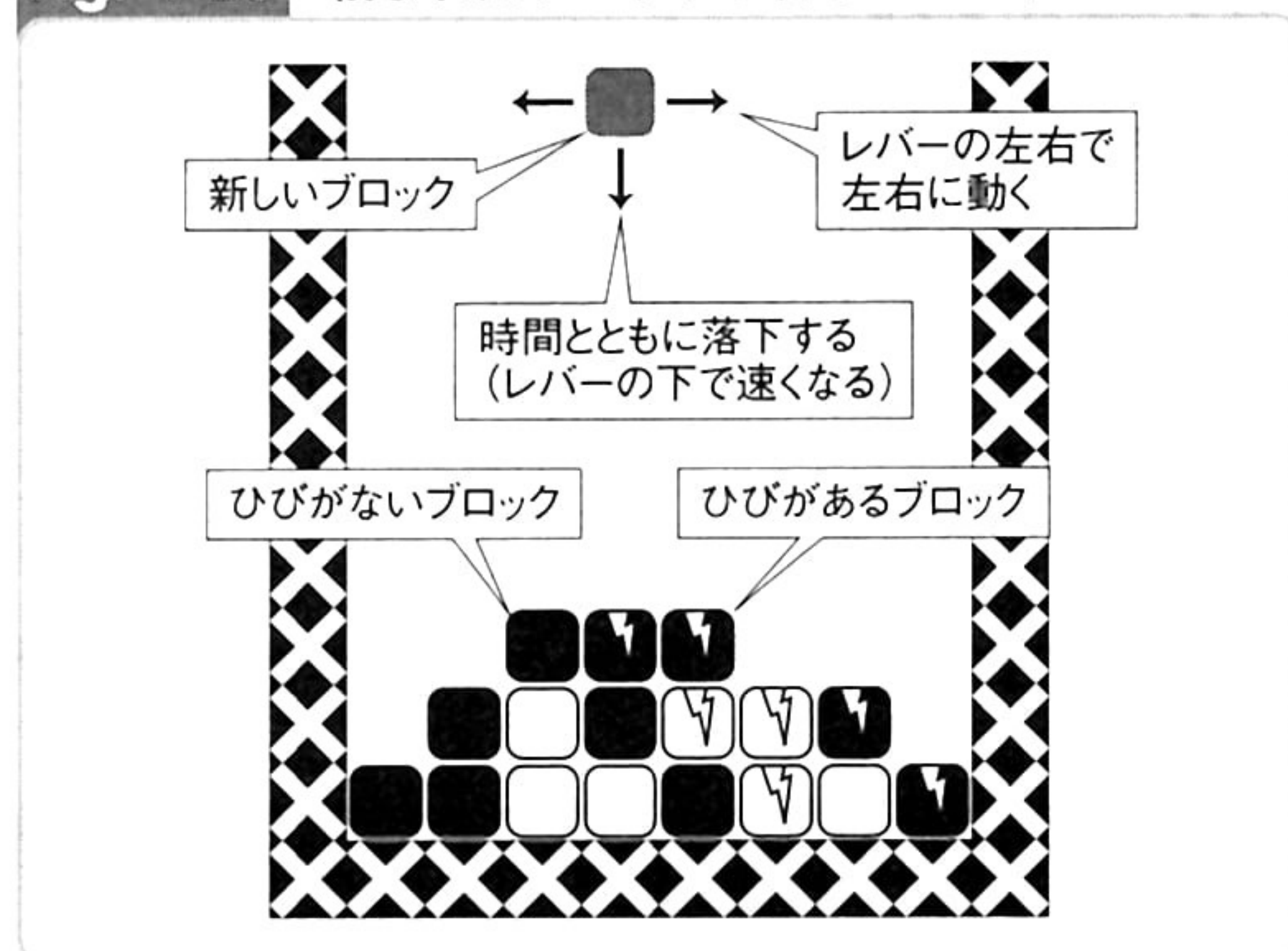



Fig. 4-24 ブロックを壊す

Fig. 4-25 衝撃が広がる

アルゴリズム

ブロックをぶつけて壊すアクションを実現するには、まずステージをセルで表現します(Fig. 4-26)。ここではブロックを「0～3」の数字で表すことにします。数字とブロックの種類の対応は、

0：黒いブロック（ひびなし）
1：白いブロック（ひびなし）
2：黒いブロック（ひびあり）
3：白いブロック（ひびあり）

となっています。実際のゲームでは、黒と白だけではなく、ブロックの色をもっと増やすとよいでしょう。その場合も、ブロックを数字で表す手法は変わりません。

ステージ上方には新しいブロックを出現させます。新しいブロックも、積まれたブロックと同様に、種類に応じた数字で表します。

新しいブロックは、時間とともに少しずつ落下します(Fig. 4-27)。ブロックが他のブロックに接触したら、ブロックをその位置に固定します。

Fig. 4-26 ステージをセルで表現する

壁（＝）
新しいブロック
黒いブロック（ひびなし）
黒いブロック（ひびあり）
白いブロック（ひびなし）
白いブロック（ひびあり）

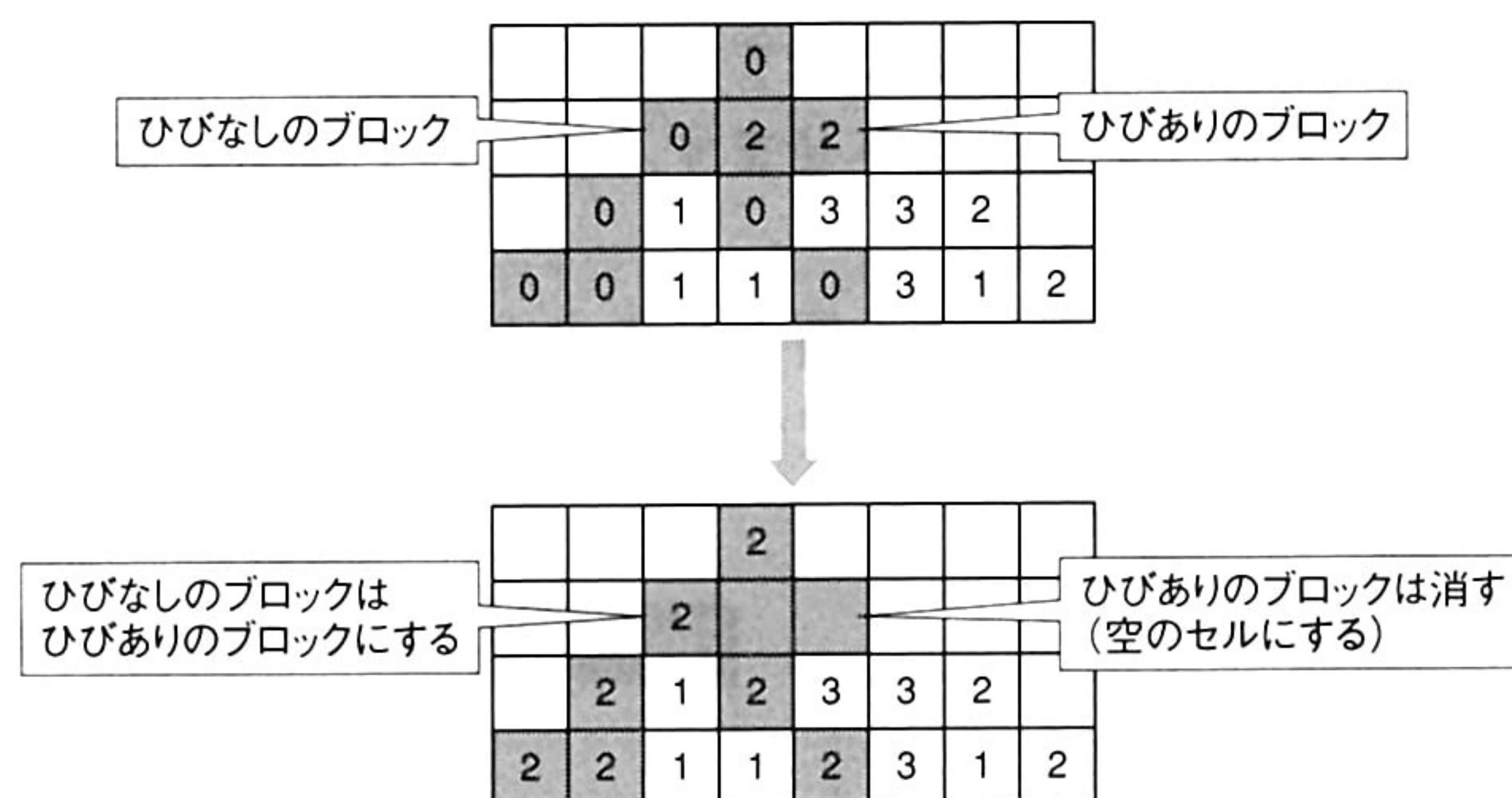
Fig. 4-27 新しいブロックの落下

他のブロックに接触したら新しいブロックを固定する
新しいブロックを時間とともに落下させる

衝撃を広げる

新しいブロックと接触したブロックが同じ種類ならば、ブロックを壊すことができます。ぶつかった衝撃は、縦・横・斜めに広がります(Fig. 4-28)。あるブロックに隣接するブロックを調べ、次に隣接するブロックについてもさらに隣接するブロックを調べ、といった具合に、再帰的に衝撃を広げていきます。同じ種類のブロックが隣接しているかぎり、処理を続けます。これは「連鎖的に消す」(→p. 106)に似た処理です。

Fig. 4-28 衝撃を広げる

Fig. 4-29 ブロックを破壊する

衝撃が及んだブロックは、ひびを入れたり、壊したりします（Fig. 4-29）。ひびなしのブロックは、ひびありのブロックに変化させ、ひびありのブロックは消去します。ブロックを消去したら、上にあったブロックを落下させます。

プログラム

List 4-3はブロックをぶつけて壊すプログラムです。ステージの移動処理と、ブロックをぶつける処理を掲載しました。

移動処理（Move関数）は初期状態・入力状態・破壊状態・落下状態に分かれています。初期状態では、新しいブロックを出現させて、入力状態に移行します。ブロックの種類はランダムに決めます。

入力状態では、レバー入力に応じて、新しいブロックを左右に動かします。また、時間とともに少しずつブロックを落下させます。落下したブロックが他のブロックに接触したら、ブロックを固定し、ブロックをぶつける処理（Strike関数）を呼び出します。

ブロックをぶつける処理では、隣接するブロックが同じ種類のブロックかどうかを調べます。

同じ種類ならば、ブロックにマークを付けて、さらに隣接するブロックについても同様に調べます。このプログラムでは、ブロックのセルにマークを付けるために、ビット演算を使っています。

マークされたブロックがある場合には、破壊状態に移行します。ここでは一定時間が経過するのを待ってから、マークされたブロックにひびを入れたり、消したりします。ブロックがひびなしならばひびを入れて、ひびありならば消します。消えたブロックがある場合には、落下状態に移行します。

落下状態では、消えたブロックの上にあったブロックを、1段ずつ下に落とします。そして、初期状態に戻り、新しいブロックを出現させます。

このサンプルでは行いませんが、落ちたブロックが下のブロックにぶつかった場合に、連鎖的に壊れるようにもできます。そのためには、落下状態で落ちたブロックについて、再びブロックをぶつける処理を適用します。

List 4-3 ブロックをぶつけて壊す（CStruckBlockStageクラス）

```
// ステージの移動処理
bool CStruckBlockStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 初期状態
    if (State==0) {

        // 新しいブロックの位置
        CX=xs/2;
        CY=0;

        // 新しいブロックの種類をランダムに決める
        Type=Rand.Int31()%STRUCK_BLOCK_COUNT;

        // タイマーなどを初期化し、
        // 入力状態に移行する
        DropTime=0;
        PrevDown=true;
        State=1;
    }

    // 入力状態
    if (State==1) {

        // レバー入力に応じて、新しいブロックを左右に移動させる
        if (!PrevLever) {
            if (is->Left && Cell->Get(CX-1, CY)==' ') CX--;
            if (is->Right && Cell->Get(CX+1, CY)==' ') CX++;
        }
```

```
    PrevLever=is->Left|is->Right;

    // 一定時間ごとに、新しいブロックを落下させる
    // レバーを下に入れたときには、時間待ちをせずに落下させる
    DropTime++;
    if (DropTime==60 || (!PrevDown && is->Down)) {
        DropTime=0;

        // 1段下のセルが空の場合には、
        // 新しいブロックを下に移動する
        if (Cell->Get(CX, CY+1)==' ') {
            CY++;
        } else

        // 1段下のセルが空ではない場合には、
        // 新しいブロックを固定する
        {
            // タイマーの初期化
            Time=0;

            // ブロックが消えなかったときには、
            // 初期状態に移行する
            State=0;

            // ブロックの固定
            Cell->Set(CX, CY, '0'+Type);

            // ブロックをぶつける処理を呼び出す
            Strike(CX, CY+1, '0'+Type);
        }
    }
    if (!is->Down) PrevDown=false;
}

// 破壊状態
if (State==2) {

    // 一定時間が経過したら、
    // 壊すブロックにひびを入れたり、消したりする
    Time++;
    if (Time==30) {

        // 消えるブロックがなければ、
        // 初期状態に移行する
        State=0;

        // 壊すブロックを探す
        for (int y=0; y<ys; y++) {
            for (int x=0; x<xs; x++) {
                char c=Cell->Get(x, y);
```

```
                //壊すブロックが見つかったら、
                //ひびを入れたり、消したりする
                if (c&0x80) {
                    c&=0x7f;

                    //ひびが入っていなければ、
                    //ブロックにひびを入れる
                    if (
                        '0'<=c &&
                        c<'0'+STRUCK_BLOCK_COUNT
                    ) {
                        Cell->Set(x, y,
                            c+STRUCK_BLOCK_COUNT);
                    } else

                    //ひびが入っていれば、
                    //ブロックを消す
                    {
                        Cell->Set(x, y, ' ');

                        //落下状態に移行する
                        Time=0;
                        State=3;
                    }
                }
            }
        }
    }
}

//落下状態
if (State==3) {

    //一定時間が経過したら、
    //消えたブロックの上にあったブロックを落下させる
    Time++;
    if (Time==30) {

        //空のセルを探す
        for (int x=1; x<xs-1; x++) {
            for (int i=1; i<ys-1; i++) {

                //空のセルが見つかったら、
                //上にあるすべてのセルを1段ずつ下に移動する
                if (Cell->Get(x, i)==' ') {
                    for (int y=i; y>0; y--) {
                        Cell->Set(x, y, Cell->Get(x, y-1));
                    }
```

```
                        // 最上段のセルは空にする
                        Cell->Set(x, 0, ' ');
                    }
                }
            }

            // 初期状態に移行する
            State=0;
        }
    }

    return true;
}

// ブロックをぶつける処理
void CStruckBlockStage::Strike(int x, int y, char c) {

  // 現在位置のセルを取得する
    char d=Cell->Get(x, y);

    // 現在位置のセルが、
    // 指定された種類のブロック（ひびあり・ひびなし）のときの処理
    if (d==c || d==c+STRUCK_BLOCK_COUNT) {

        // ブロックを壊すブロックにする
        // ビット演算を使って、ブロックにマークを付ける
        Cell->Set(x, y, d|0x80);

        // 上下左右斜め（8方向）のブロックについても、
        // ブロックをぶつける処理を再帰的に行う
        for (int i=-1; i<=1; i++) {
            for (int j=-1; j<=1; j++) {
                if (i!=0 || j!=0) {
                    Strike(x+i, y+j, c);
                }
            }
        }

        // 破壊状態に移行する
        State=2;
    }
}
```

SAMPLE

「STRUCK BLOCK」は「ブロックをぶつけて壊す」のサンプルです。

レバーの左右（カーソルキーの左右）でブロックが移動します。ブロックは時間とともに少しずつ落下します。レバーの下（カーソルキーの下）で落下スピードが上がります。

ブロックを同じ種類のブロックの上に落とすと、ブロックにひびを入れたり、消したりすることができます。

ひびなしのブロックにはひびが入り、ひびありのブロックは壊れて消えます。縦・横・斜めに隣接するブロックも、同様にひびが入ったり、消えたりします。ブロックを消すと、上にあったブロックが落下します。

STRUCK BLOCK → p. 388

ステージを回転させる

ステージを左右に90度ずつ回転させるアクションです。ステージの向きを変えることによって、ブロックの落下方向を変えることができます。

ステージにはブロックが積まれています(Fig. 4-30)。レバーを左に入力すると、ステージが左に90度回転します(Fig. 4-31)。このとき、ブロックは新しく下になった方向に落下します。

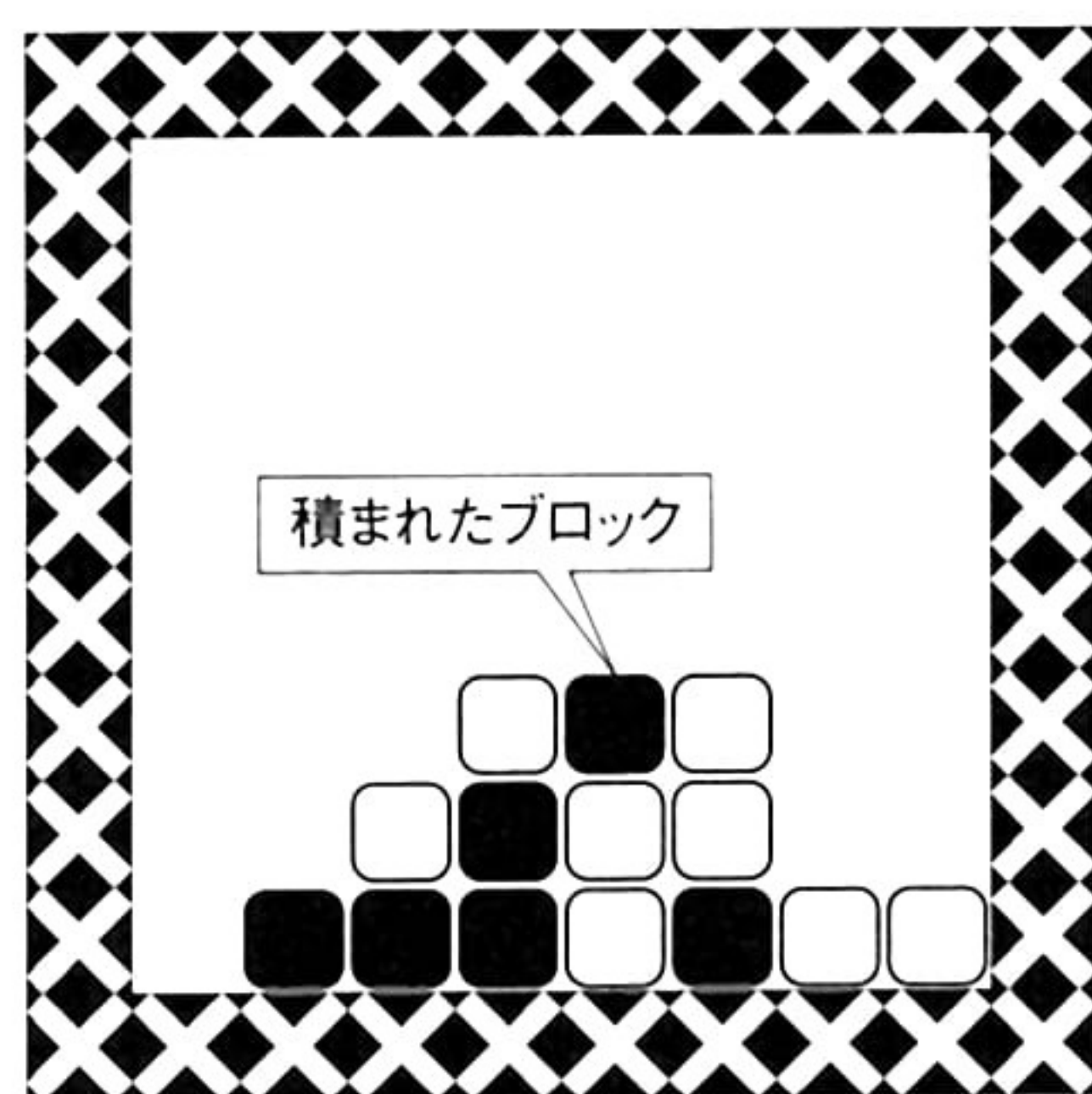
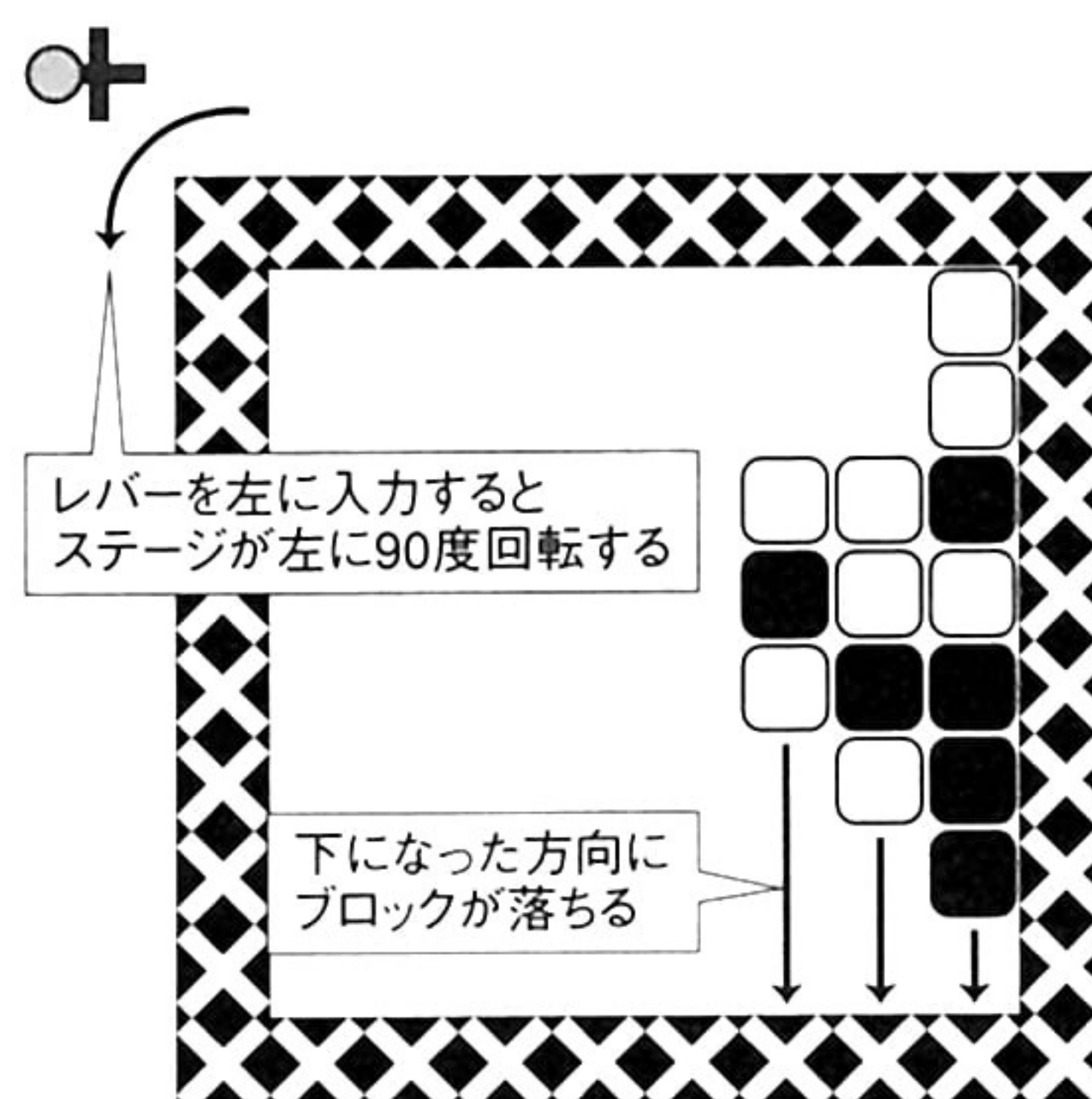
レバーを右に入力したときも、同様にステージが右に90度回転します。この場合も、ブロックは新しく下になった方向に落ちます。

ステージを回転させるゲームには『ぐるりん』があります。このゲームはいわゆる「落ち物パズルゲーム」の一種ですが、ステージを回転させられることが特徴です。回転によって落下方向が変わり、積まれたブロックの配置が変化するので、今までは消せなかったブロックを消すことができます。また、ステージが滑らかに回転するので、見た目にも楽しめます。

Fig. 4-30 ステージに積まれたブロック

積まれたブロック

Fig. 4-31 ステージを左に回転させる

レバーを左に入力すると
ステージが左に90度回転する

下になった方向に
ブロックが落ちる

アルゴリズム

ステージを回転させるには、まずステージをセルで表現します(Fig. 4-32)。ここではブロックを「0〜1」の数字で表しました。縦横のセルの個数は同じにしておきます。なお、本書のサ

Fig. 4-32 ステージをセルで表現する

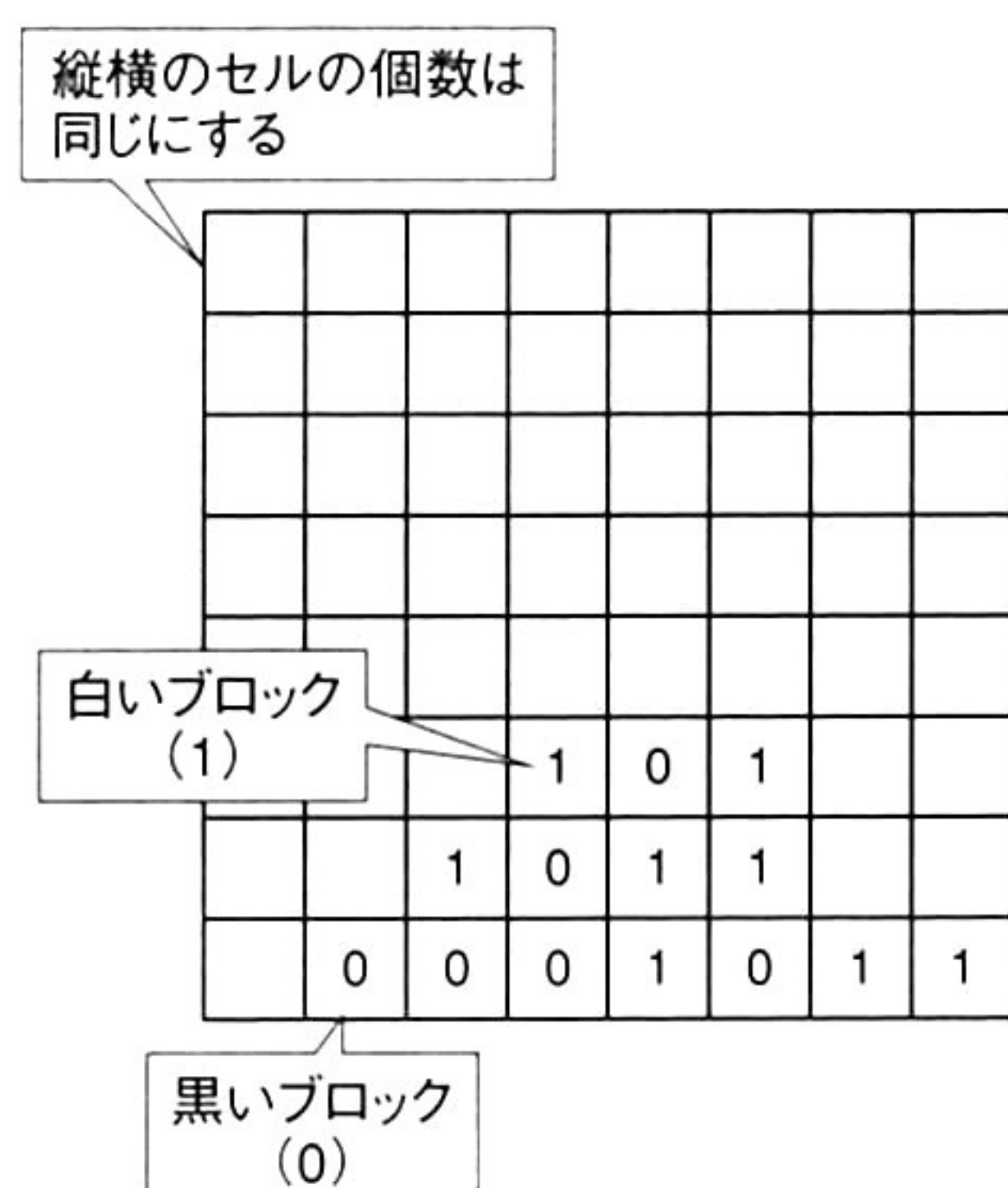



Fig. 4-33 四隅のセルを回転させる

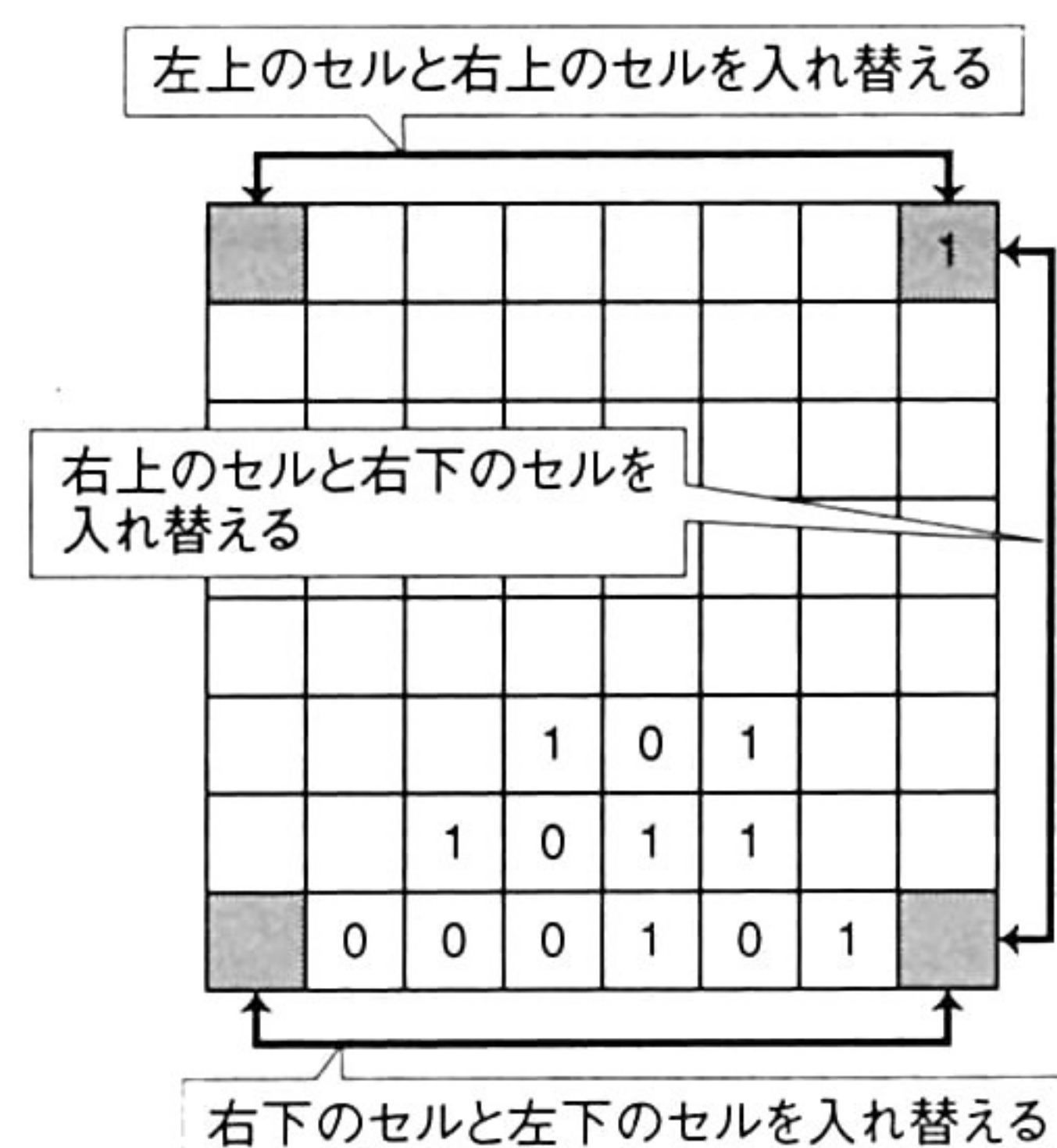



Fig. 4-34 隣のセルを回転させる

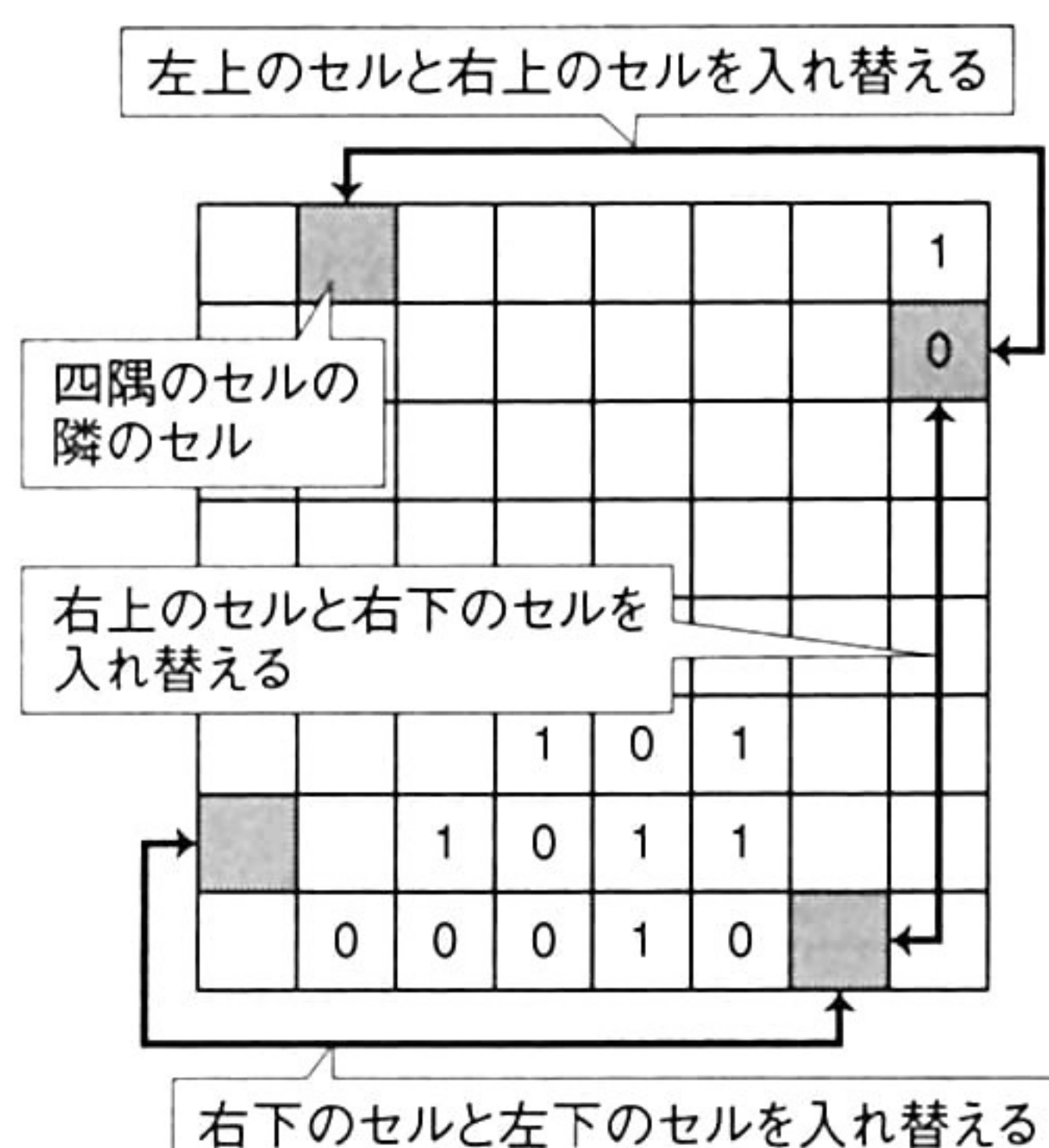
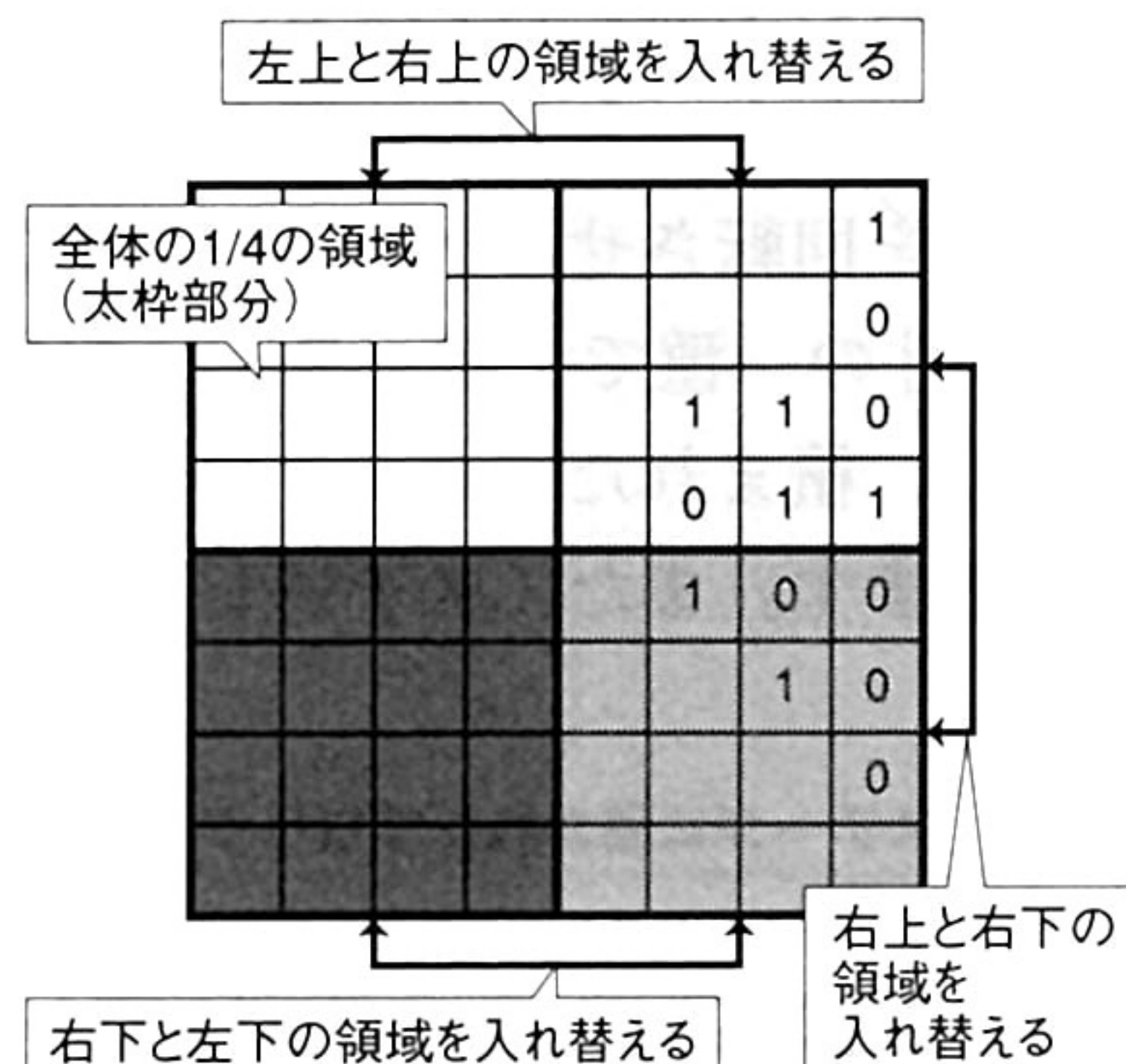



Fig. 4-35 ステージ全体を回転させる

ンプルではステージの周囲に壁がありますが、図では省略しています。

ステージを左に回転させることを考えましょう。まず、四隅のセルを回転後の状態にします。左上のセルを左下に、右上のセルを左上に、右下のセルを右上に、左下のセルを右下に移動します。これは、

左上と右上
右上と右下
右下と左下

のセルをそれぞれ入れ替えても、同じ結果になります(Fig. 4-33)。

四隅のセルの隣のセルについても、同様に入れ替えます(Fig. 4-34)。その隣のセルについても同様です。同じ要領で、全体の1/4の領域についてセルを入れ替えれば、ステージ全体が回転します(Fig. 4-35)。

右に回転させるときも、まったく同様です。セルを入れ替える順番を逆にして、

右下と左下
右上と右下
左上と右上

の順に入れ替えれば、回転方向が逆になります。

プログラム

List 4-4はステージを回転させるプログラムです。ステージの移動処理を掲載しました。

移動処理は入力状態と落下状態に分かれています。入力状態では、レバーを左に入力したらステージを左に回転させ、右に入力したら右に回転させます。ステージの1/4について、セルを入れ替えることによって、ステージ全体を回転させることができます。

ステージを回転させたら、落下状態に移行します。落下状態では、一定時間が経過してから、下に何もないブロックを落下させます。

このプログラムはステージを回転させるだけです。実際のゲームでは、ブロックを落としたり、ブロックを並べて消したりといった処理を組み合わせるとよいでしょう。

List 4-4 ステージを回転させる(CRotatedStageStageクラス)

```
// ステージの移動処理
bool CRotatedStageStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // ステージ内部の左上の座標、ステージ内部のサイズ
    int rx=4, ry=2, rs=8;

    // 入力状態
    if (State==0) {

        // レバーを左に入力したら、ステージを左に回転させる
        if (!PrevLever && is->Left) {

            // ステージの1/4の領域について、セルを入れ替える
            for (int y=0; y<rs/2; y++) {
                for (int x=0; x<rs/2; x++) {
                    Cell->Swap(rx+x, ry+y, rx+rs-1-y, ry+x);
                    Cell->Swap(rx+rs-1-y, ry+x, rx+rs-1-x, ry+rs-1-y);
                    Cell->Swap(rx+rs-1-x, ry+rs-1-y, rx+y, ry+rs-1-x);
                }
            }

            // 落下状態に移行する
```

```
            Time=0;
            State=1;
        } else

        // レバーを右に入力したら、ステージを右に回転させる
        if (!PrevLever && is->Right) {

            // ステージの1/4の領域について、セルを入れ替える
            //（左回転とは逆の順番でセルを入れ替える）
            for (int y=0; y<rs/2; y++) {
                for (int x=0; x<rs/2; x++) {
                    Cell->Swap(rx+rs-1-x, ry+rs-1-y, rx+y, ry+rs-1-x);
                    Cell->Swap(rx+rs-1-y, ry+x, rx+rs-1-x, ry+rs-1-y);
                    Cell->Swap(rx+x, ry+y, rx+rs-1-y, ry+x);
                }
            }

            // 落下状態に移行する
            Time=0;
            State=1;
        }

        // レバーを入れっぱなしにしたときに、
        // 連続して回転させないための処理
        PrevLever=is->Left||is->Right;
    }

    // 落下状態
    if (State==1) {

        // 一定時間が経過したら、ブロックを落下させる
        Time++;
        if (Time==30) {

            // 空のセルを探す
            for (int x=rx; x<rx+rs; x++) {
                for (int y=ry; y<ry+rs; y++) {

                    // 空のセルがあったら、上にあるセルを1段ずつ落下させる
                    if (Cell->Get(x, y)==' ') {
                        for (int i=y; i>ry; i--) {
                            Cell->Set(x, i, Cell->Get(x, i-1));
                        }

                        // 最上段のセルは空にする
                        Cell->Set(x, ry, ' ');
                    }
                }
            }
```

```
            // 入力状態に移行する
            State=0;
        }
    }

    return true;
}
```

SAMPLE

「ROTATED STAGE」は「ステージを回転させる」のサンプルです。レバーの左右（カーソルキーの左右）で、ステージが左右に回転します。ステージが回転すると、ブロックも一緒に回転します。回転すると落下の方向が変わるので、ブロックは新しい落下方向に落ちます。

ROTATED STAGE → p. 388

エサのブロックを消す

エサのブロックと動物のブロックを隣接させると、動物がエサを食べて、エサのブロックを消すことができるというアクションです。エサと動物には種類があり、それぞれの動物が好きなエサ、つまり消せるエサが決まっています。

ステージにはエサと動物のブロックが積まれています（Fig. 4-36）。ここではエサと動物を2種類ずつ用意しました。ペンギンは魚を食べ、クマはキノコを食べます。

ステージ上方からは新しいブロックが降ってきます。ブロックはエサまたは動物で、種類はランダムに決まります。

Fig. 4-36 エサと動物のブロック

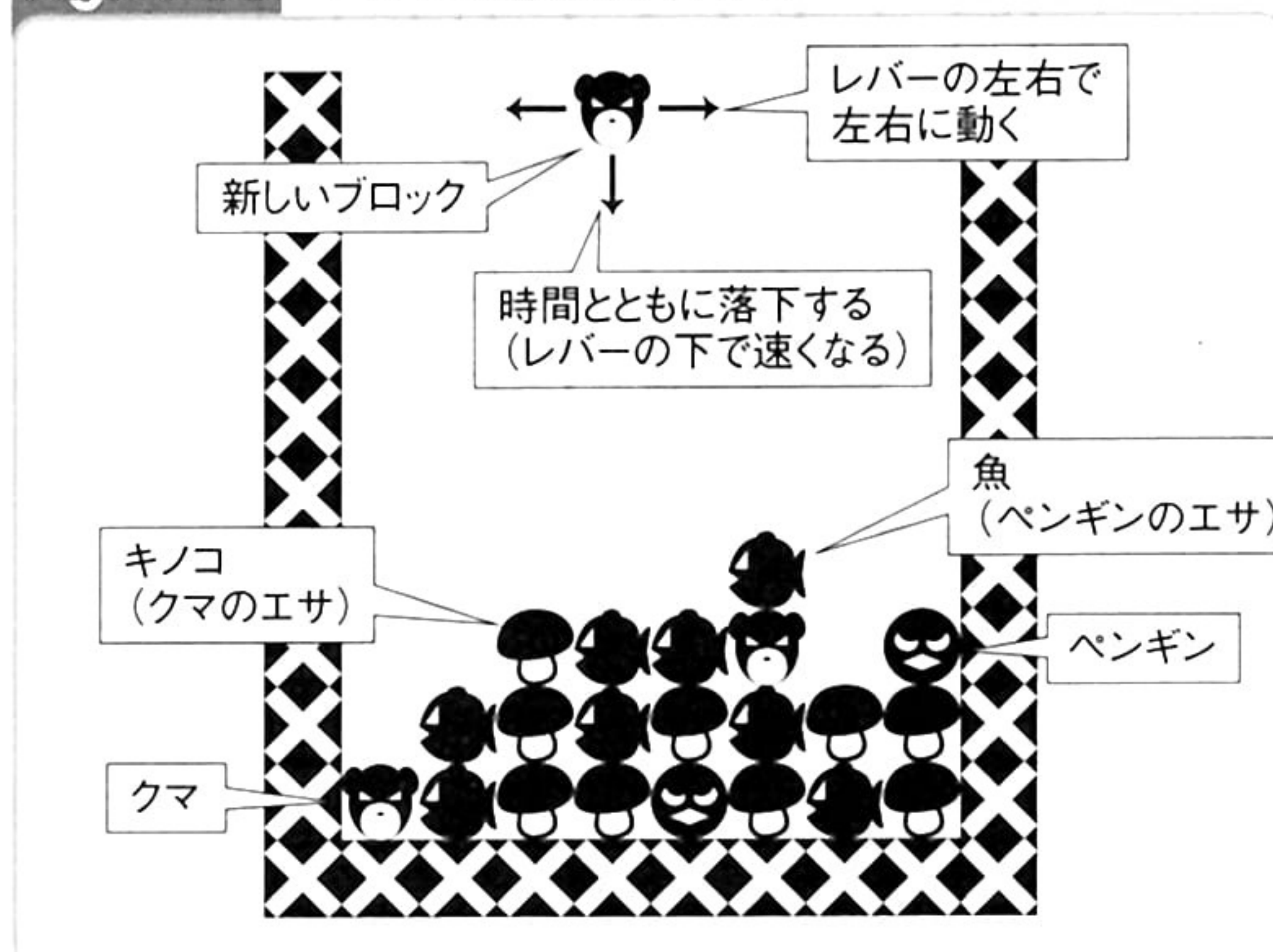
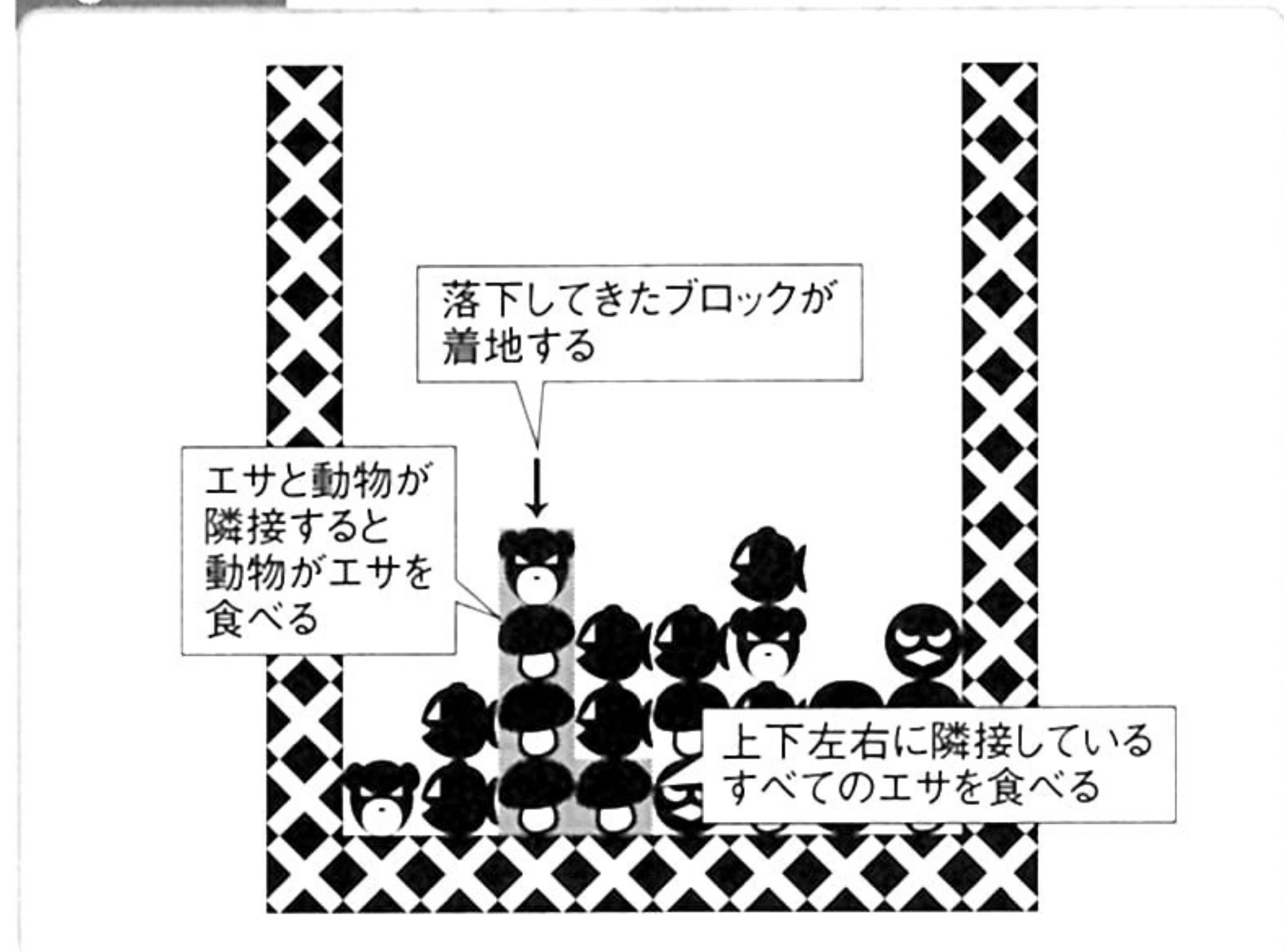



Fig. 4-37 動物がエサを食べる

新しいブロックを積んだときに、エサと動物が隣接すると、動物がエサを食べます(Fig. 4-37)。ただし、動物は自分が好きなエサしか食べません。エサを食べると、エサと動物のブロックが消えます。同じ種類のエサが上下左右に隣接している場合、動物は隣接したすべてのエサを食べます。ブロックが消えると、上にあったブロックが落ちてきます。このとき再びエサと動物のブロックが隣接すると、連鎖的にブロックを消すことができます。

エサのブロックを消すアクションは『ぱくぱくアニマル』に採用されています。このゲームにはイヌ・ウサギ・サル・パンダなどが登場し、それぞれ骨・にんじん・バナナ・笹を食べます。ブロックを消すときには、動物が動き回ってエサを食べる演出があるなど、見た目にも楽しいゲームです。

エサのブロックを消すのに似たアクションは、他のゲームにも採用されています。『スーパーパズルファイターⅡX』では、宝石に同じ色の特別な宝石を隣接させると、宝石を壊して消すことができます。『コズモギャング・ザ・パズル』では、積み上げた敵にボールを落とすと、敵を弾き飛ばして消すことができます。

アルゴリズム

エサのブロックを消すアクションは、「連鎖的に消す」(→p. 106)と似た方法で実現できます。「連鎖的に消す」では、同じ種類のボールが隣接した場合に消しますが、エサのブロックを消す場合には、動物とエサが隣接した場合に消します。

まず、ステージをセルで表現します(Fig. 4-38)。ステージの壁は「=」で、ブロックは「0〜3」の数字で表すことにしました。数字とブロックの種類は、

0：エサ(魚)
1：エサ(キノコ)

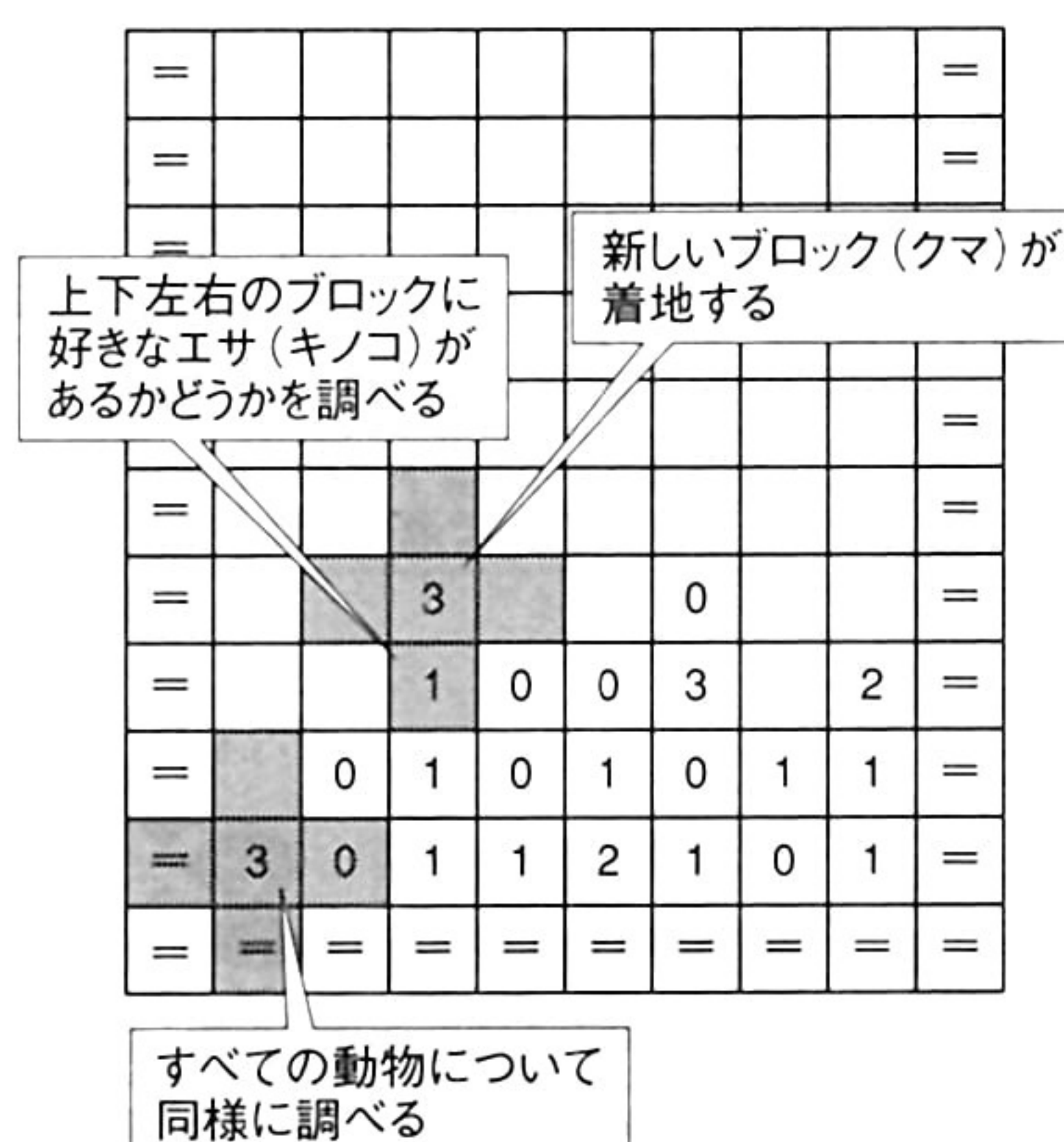
Fig. 4-38 ステージをセルで表現する

=				3					=
=									=
=									=
=									=
=									=
=									=
=						0			=
			1	0	0	3		2	=
=		0	1	0	1	0	1	1	=
=	3	0	1	1	2	1	0	1	=
=	=	=	=	=	=	=	=	=	=

Fig. 4-39 動物とエサの隣接を調べる

=									=
=									=
=									
									=
=									=
=			3			0			=
=			1	0	0	3		2	=
=		0	1	0	1	0	1	1	=
=	3	0	1	1	2	1	0	1	=
=	=	=	=	=	=	=	=	=	=

Fig. 4-40　動物とエサが隣接している場合

エサと隣接していたら
動物のブロックを
消えるブロックにする

エサのブロックも
消えるブロックにする

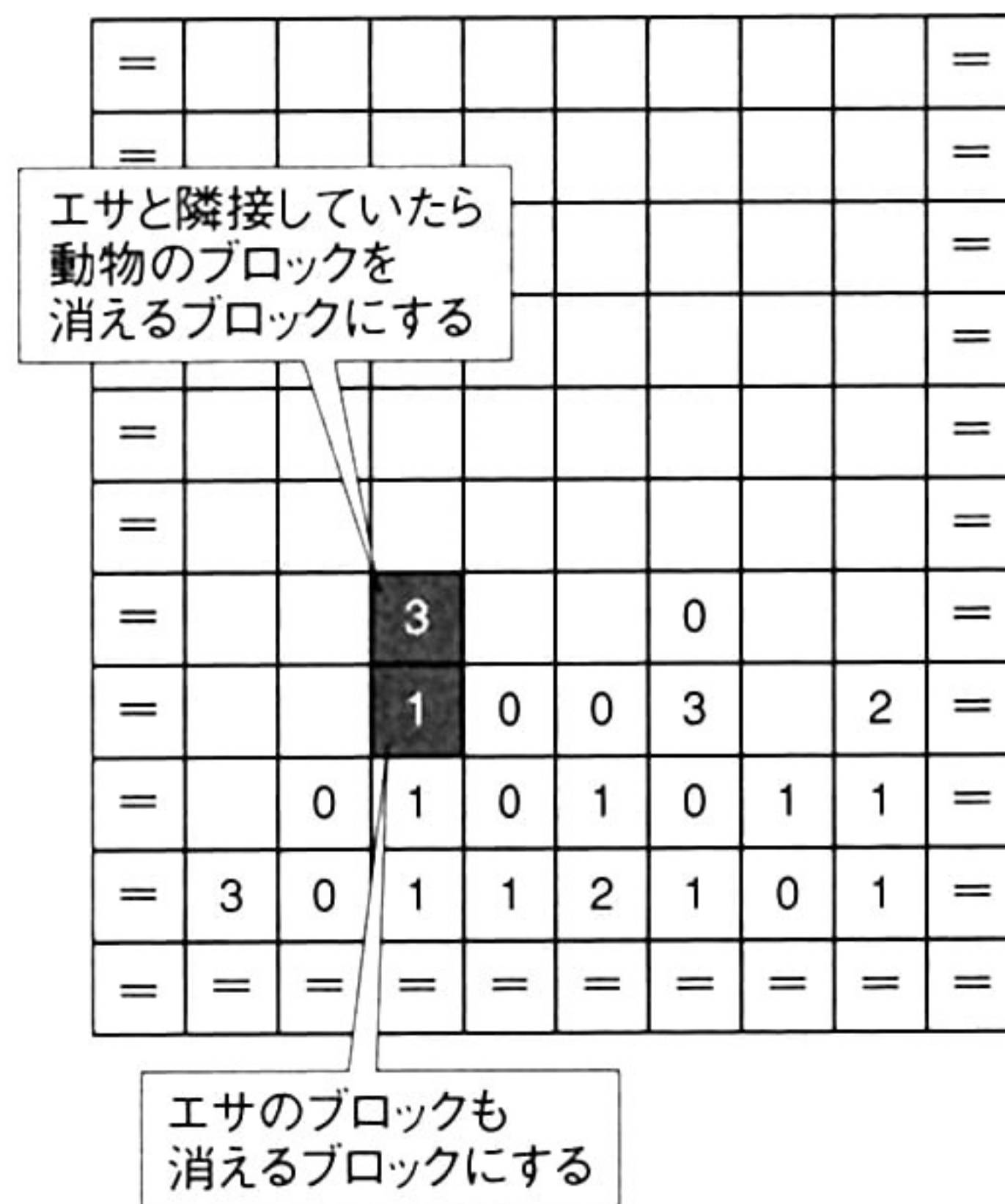
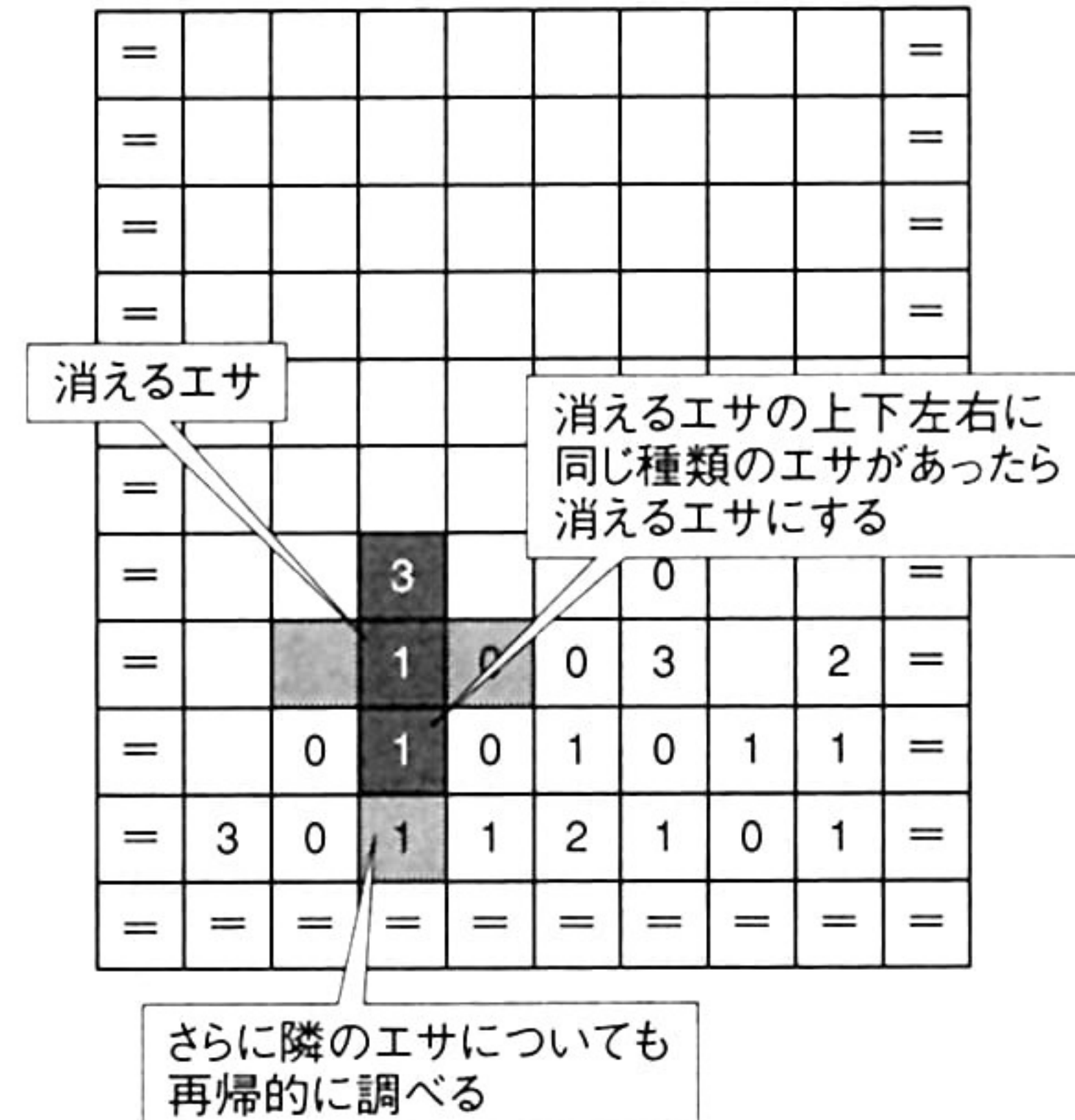



Fig. 4-41　隣接した同じ種類のエサを再帰的に調べる

消えるエサ

消えるエサの上下左右に
同じ種類のエサがあったら
消えるエサにする

さらに隣のエサについても
再帰的に調べる

2：動物（ペンギン）
3：動物（クマ）

のように対応しています。

ステージ上方からは新しいブロックが降ってきます。新しいブロックが他のブロックに着地したら、動物とエサが隣接しているかどうかを調べます（Fig. 4-39）。ステージ内のすべての動物について、上下左右に好きなエサがあるかどうかを調べます。

動物が好きなエサと隣接していたら、動物とエサのブロックを、消えるブロックにします（Fig. 4-40）。消えるエサについては、上下左右のセルも再帰的に調べて、同じ種類のエサが隣接している場合には、消えるブロックにします（Fig. 4-41）。

消えるブロックは、一定時間が経過したら、空のセルにします。空になったセルの上にあるセルは、1段ずつ落下させます。そして、再び動物とエサの隣接を調べ、もしも隣接していたら、連鎖的にブロックを消します。

プログラム

List 4-5はエサのブロックを消すプログラムです。ステージの移動処理と、エサを食べる処理を掲載しました。

移動処理（Move関数）は、初期状態・入力状態・消去判定状態・消去状態・落下状態に分かれています。これは「連鎖的に消す」（→p. 106）に似た構成です。

初期状態では、新しいブロックを出現させ、入力状態に移行します。ブロックの種類はランダムに決めます。

入力状態では、レバー入力に応じて、新しいブロックを左右に動かします。また、時間とともに少しずつブロックを落下させます。ブロックが着地したら、消去判定状態に移行します。

消去判定状態では、動物のセルを探し、周囲にエサのセルがあるかどうかを調べます。エサがある場合には、動物とエサを消えるブロックとしてマークし、エサを食べる処理（Eat関数）を呼び出します。マークにはビット演算を使いました。

エサを食べる処理では、同じ種類のエサが隣接するかぎり、消えるブロックとしてマークします。上下左右のセルを再帰的に調べることによって、連続する同じ種類のエサをすべて消えるブロックにします。

消えるエサがある場合には、消去状態に移行します。消去状態では、一定時間が経過したら、消えるブロックを完全に消します。消えるブロックのセルを空にして、落下状態に移行します。

落下状態では、空になったセルの上にあるセルを、1段ずつ落下させます。そして、消去判定状態に移行します。消去判定状態に戻るのは、落下後に動物とエサが隣接した場合に、連鎖的にブロックを消すためです。

List 4-5 エサのブロックを消す（CFoodBlockStageクラス）

```
// ステージの移動処理
bool CFoodBlockStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 初期状態
    if (State==0) {

        // 新しいブロックの位置
        CX=xs/2;
        CY=0;

        // 新しいブロックの種類をランダムに決める
        Type=Rand.Int31()%(FOOD_BLOCK_COUNT*2);

        // タイマーなどを初期化し、入力状態に移行する
        DropTime=0;
        PrevDown=true;
        State=1;
    }

    // 入力状態
    if (State==1) {

        // レバー入力に応じて、新しいブロックを左右に動かす
        if (!PrevLever) {
            if (is->Left && Cell->Get(CX-1, CY)==' ') CX--;
            if (is->Right && Cell->Get(CX+1, CY)==' ') CX++;
        }
        PrevLever=is->Left||is->Right;

        // 一定時間ごとに、新しいブロックを落下させる
```

```
        DropTime++;
        if (DropTime==60 || (!PrevDown && is->Down)) {
            DropTime=0;

            // 下のセルが空ならば、落下させる
            if (Cell->Get(CX, CY+1)==' ') {
                CY++;
            } else

            // 下のセルが空でなければ、
            // ブロックを固定し、消去判定状態に移行する
            {
                Cell->Set(CX, CY, '0'+Type);
                State=2;
            }
        }
        if (!is->Down) PrevDown=false;
    }

    // 消去判定状態
    if (State==2) {

        // タイマーの初期化
        Time=0;

        // ブロックが消えなかったときには、初期状態に移行する
        State=0;

        // 動物のセルを探す
        for (int y=0; y<ys; y++) {
            for (int x=0; x<xs; x++) {
                char eater=Cell->Get(x, y);

                // 動物のセルが見つかったら、
                // 周囲にあるエサを食べさせる
                if (
                    '0'+FOOD_BLOCK_COUNT<=eater &&
                    eater<'0'+FOOD_BLOCK_COUNT*2
                ) {
                    char food=eater-FOOD_BLOCK_COUNT;

                    // 動物の周囲にエサがあったら、
                    // エサを食べる処理を呼び出す
                    if (
                        (Cell->Get(x-1, y)&0x7f)==food ||
                        (Cell->Get(x+1, y)&0x7f)==food ||
                        (Cell->Get(x, y-1)&0x7f)==food ||
                        (Cell->Get(x, y+1)&0x7f)==food
                    ) {
                        // 動物を消えるブロックとしてマークする
```

```
                        Cell->Set(x, y, eater|0x80);

                        //周囲のエサを食べさせる
                        Eat(x-1, y, food);
                        Eat(x+1, y, food);
                        Eat(x, y-1, food);
                        Eat(x, y+1, food);

                        //消去状態に移行する
                        State=3;
                    }
                }
            }
        }
    }

    //消去状態
    if (State==3) {

        //一定時間が経過したら、消えるブロックを完全に消す
        Time++;
        if (Time==30) {
            for (int y=0; y<ys; y++) {
                for (int x=0; x<xs; x++) {

                    //消えるブロックを見つけたら、
                    //セルを空にする
                    if (Cell->Get(x, y)&0x80) {
                        Cell->Set(x, y, ' ');
                    }
                }
            }

            //落下状態に移行する
            Time=0;
            State=4;
        }
    }

    //落下状態
    if (State==4) {

        //一定時間が経過したら、下に何もないブロックを落下させる
        Time++;
        if (Time==30) {

            //空のセルを探す
            for (int x=0; x<xs; x++) {
                for (int i=0; i<ys; i++) {
```

```
                    //空のセルを見つけたら、
                    //上にあるセルを1段ずつ落下させる
                    if (Cell->Get(x, i)==' ') {
                        for (int y=i; y>0; y--) {
                            Cell->Set(x, y, Cell->Get(x, y-1));
                        }

                        //最上段のセルは空にする
                        Cell->Set(x, 0, ' ');
                    }
                }
            }

            //消去判定状態に移行する
            State=2;
        }
    }

    return true;
}

//エサを食べる処理
void CFoodBlockStage::Eat(int x, int y, char food) {

    //現在位置のセルを取得する
    char c=Cell->Get(x, y);

    //セルが指定されたエサのときの処理
    if (c==food) {

        //エサを消えるブロックとしてマークする
        Cell->Set(x, y, c|0x80);

        //上下左右のセルについても、
        //エサを食べる処理を再帰的に行う
        Eat(x-1, y, food);
        Eat(x+1, y, food);
        Eat(x, y-1, food);
        Eat(x, y+1, food);
    }
}
```

SAMPLE

「FOOD BLOCK」は「エサのブロックを消す」のサンプルです。

ステージ上方からエサまたは動物のブロックが降ってきます。ブロックはレバーの左右（カーソルキーの左右）で左右に移動します。レバーの下（カーソルキーの下）を入力すると、落下スピードが上がります。

ブロックが着地したときに、エサと動物が隣接していると、エサと動物のブロックが消えます。動物とエサには種類があり、それぞれの動物が好きなエサに隣接した場合だけ、消すことができます。

ブロックを消すと、上にあったブロックが落ちてきます。このとき、再びエサと動物が隣接すると、連鎖的に消すことができます。

FOOD BLOCK → p. 388

ブロックで囲んで消す

ブロックでアイテムの周りを囲むことによって、アイテムを消すアクションです。アイテムの周囲を隙間なく囲むと、囲んだアイテムを消すことができます。

ステージにはブロックとアイテムが積まれています（Fig. 4-42）。ステージ上方からは、新しいブロックまたはアイテムが降ってきます。ブロックやアイテムは、レバー入力で左右に動かすことができます。また、レバーを下に入力すると、落下スピードを上げることができます。

新しいブロックやアイテムが着地したときに、ステージ内にブロックで囲まれたアイテムがあったら、そのアイテムは消えます（Fig. 4-43）。アイテムの上下左右に空間がなく、ブロックや壁でふさがれていたら、アイテムを囲んだことになります。

複数のアイテムを同時に囲んで消すこともできます（Fig. 4-44）。この場合は、すべてのアイテムの上下左右に空間がないときに、囲んだとみなします。アイテム同士が隣り合ってもかまいませんが、いずれのアイテムの隣にも、空間があってはいけません。

ブロックで囲んで消すアクションは『クレオパトラフォーチュン』に採用されています。このゲームでは、降ってくるブロックやアイテムに種類があり、種類ごとに形状が違います。いろいろな形のブロックを並べて、アイテムを囲むには、素早くブロックを操作するだけではなく、ブロックの置き方にも頭を使う必要があります。なお、このゲームでは、囲んだ領域内に空間があっても、囲んだとみなされます。

Fig. 4-42 ブロックとアイテム

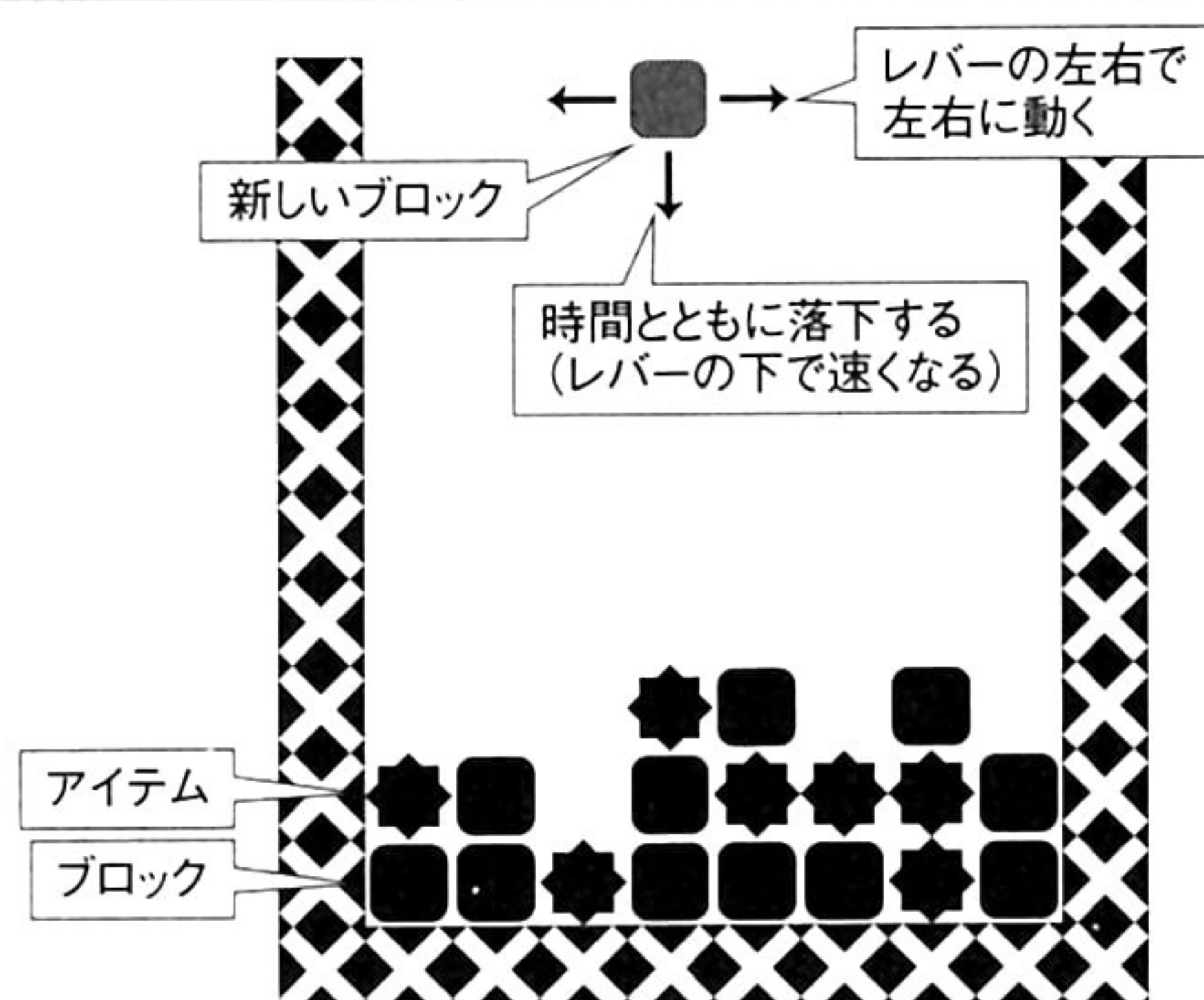



Fig. 4-43　ブロックで囲んだアイテムが消える

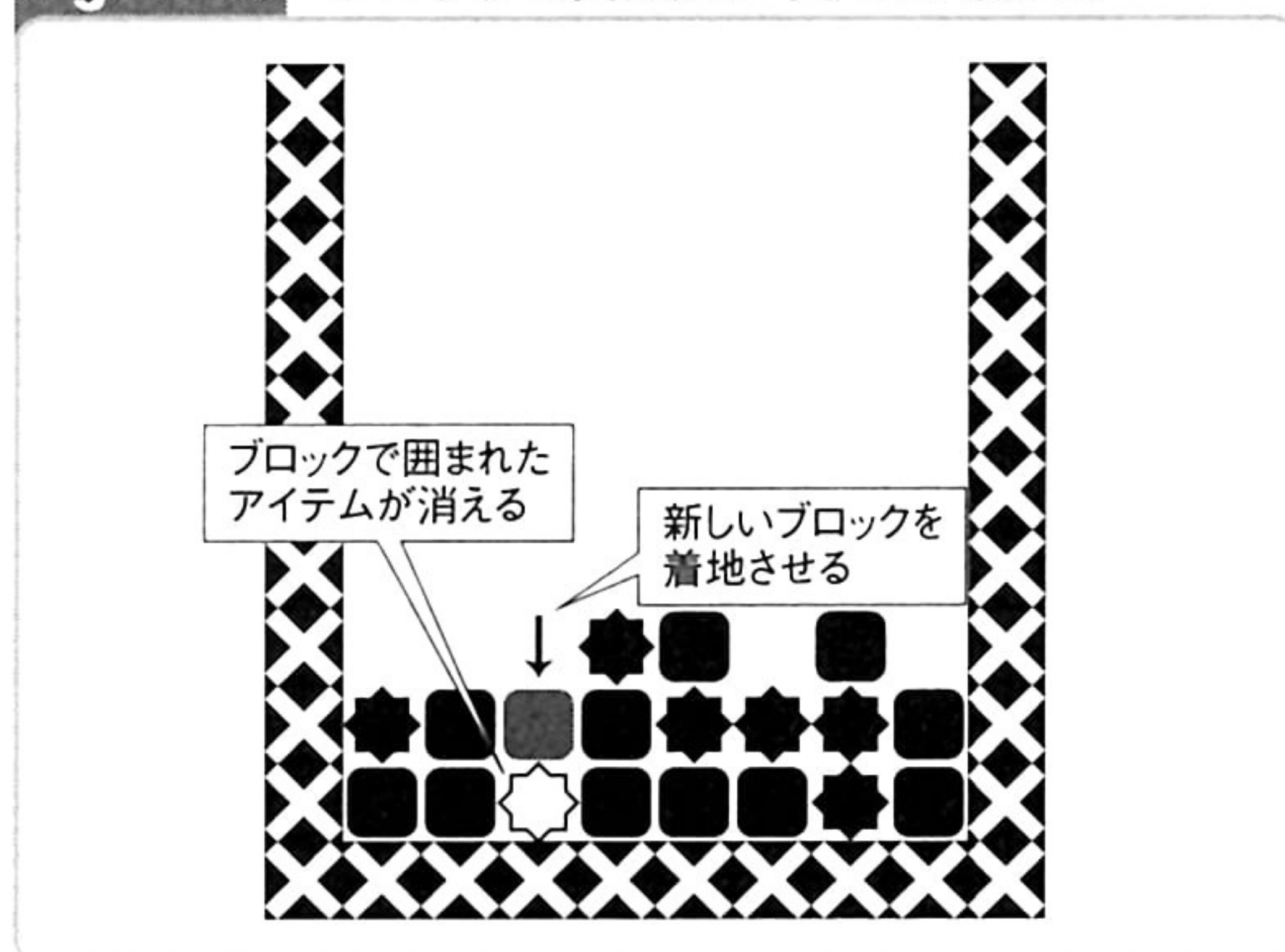
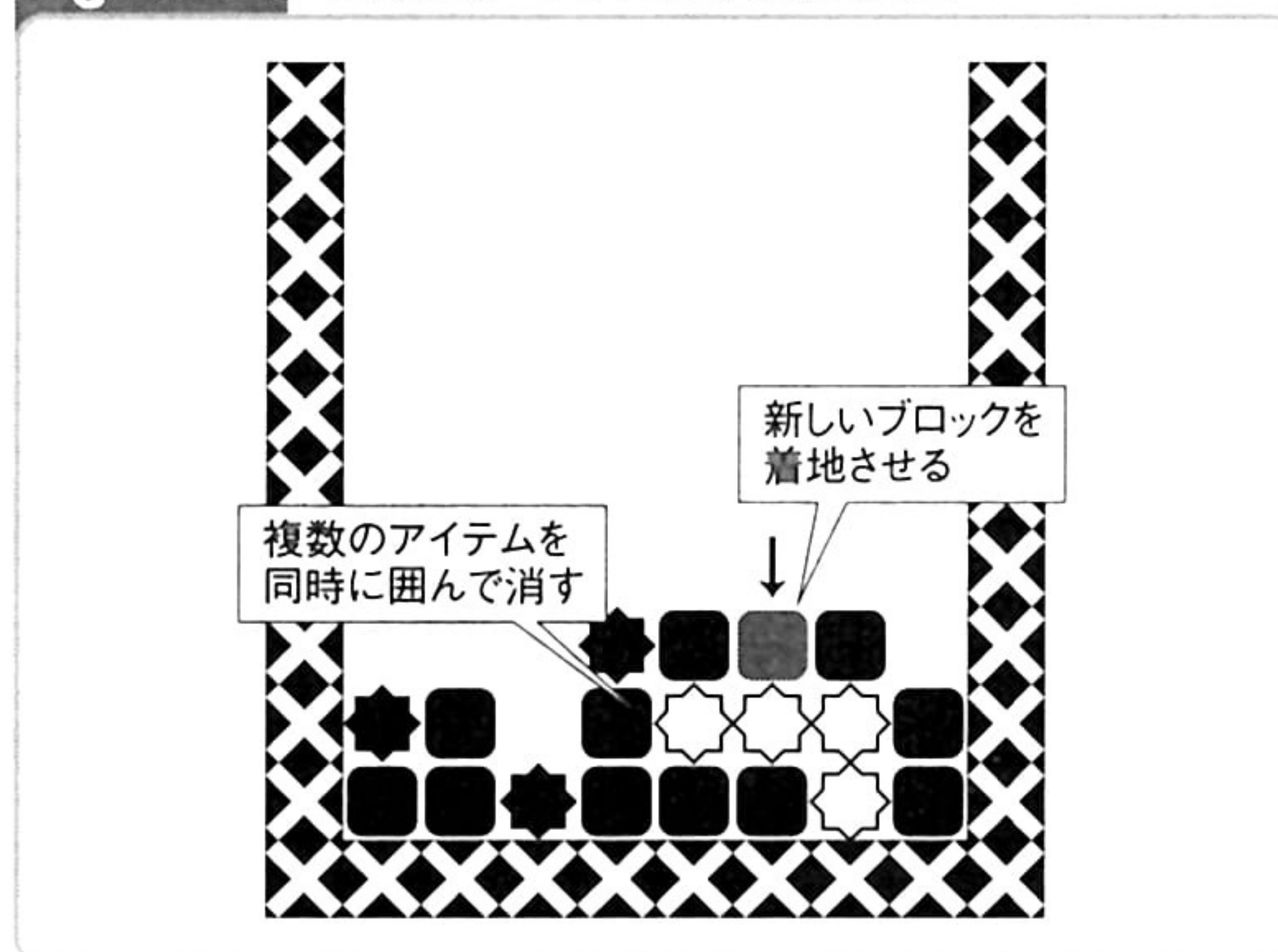



Fig. 4-44　複数のアイテムを同時に囲む

アルゴリズム

ブロックで囲んで消すアクションを実現するためには、まずステージをセルで表現します(Fig. 4-45)。ステージの壁は「=」で、ブロックは「#」で、アイテムは「+」で表しました。

ステージ上方からは新しいブロックまたはアイテムが降ってきます。ブロックやアイテムが着地したら、ステージ内に囲まれたアイテムがあるかどうかを調べます(Fig. 4-46)。

囲まれているかどうかは、アイテムの上下左右に、空のセルがあるかどうかによって判定します。空のセルがあれば、アイテムは囲まれていません。空のセルがなければ、囲まれている可能性があります。

アイテムの隣に他のアイテムがある場合には、隣のアイテムの上下左右のセルについても調べます(Fig. 4-47)。さらに隣にも別のアイテムがある場合には、そのアイテムの上下左右についても再帰的に調べます。

このように、隣接するすべてのアイテムについて、上下左右のセルを調べます。すべてのア

Fig. 4-45　ステージをセルで表現する

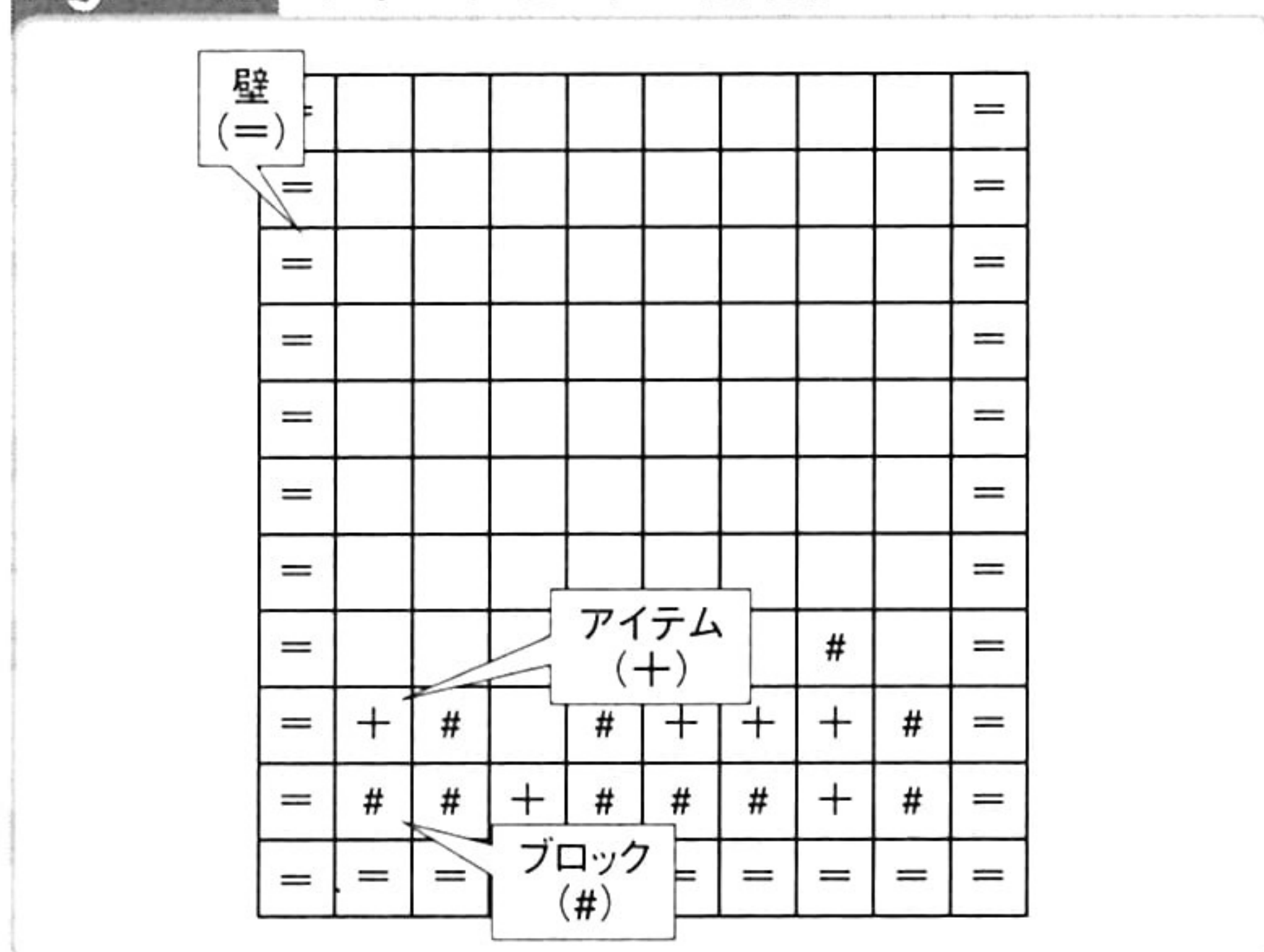



Fig. 4-46　囲まれたアイテムがあるかどうかを調べる

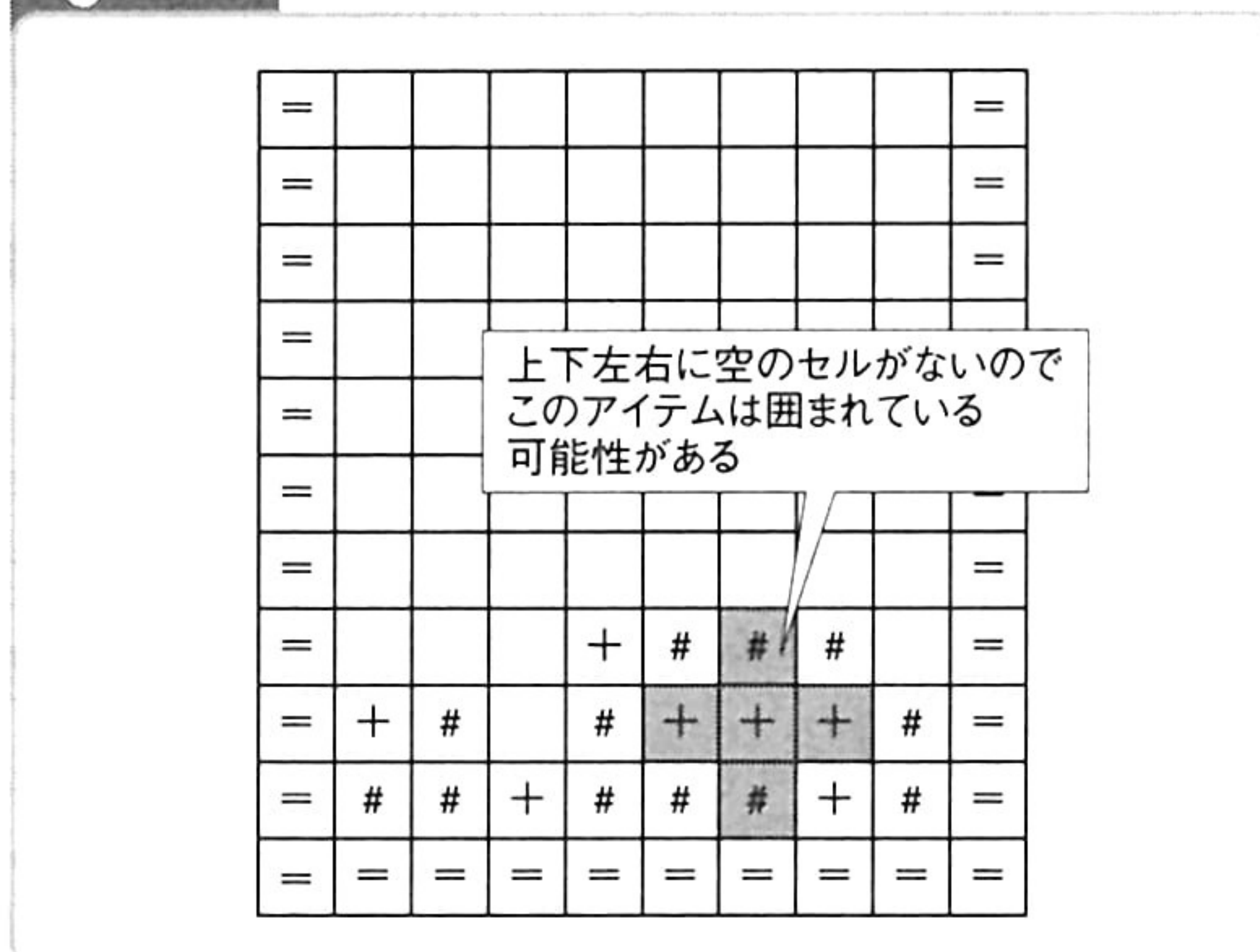



Fig. 4-47 隣のアイテムの上下左右を調べる

右隣のアイテムについても
上下左右のセルを調べる
調査ずみのアイテム
左隣のアイテムについても
上下左右のセルを調べる

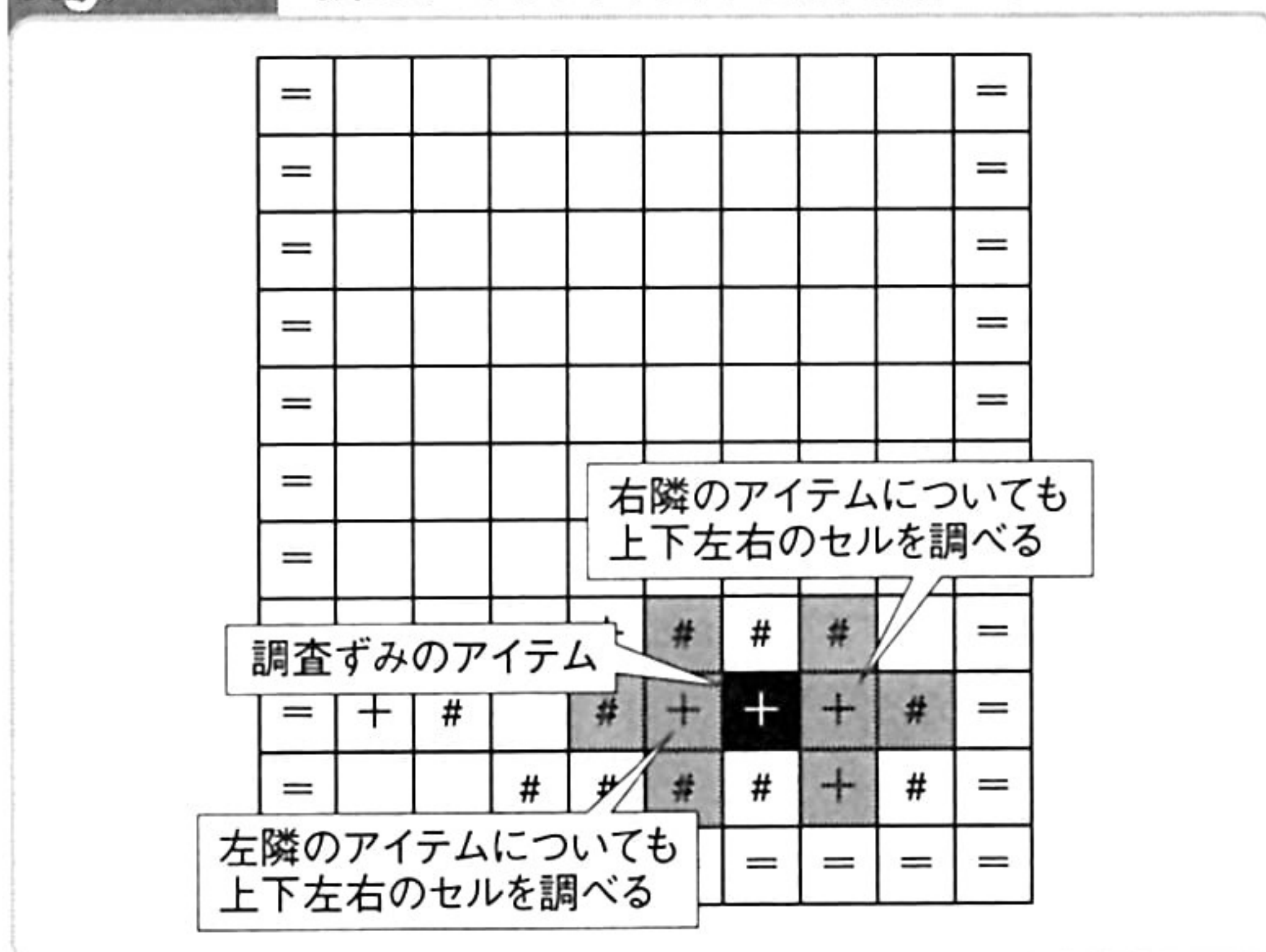
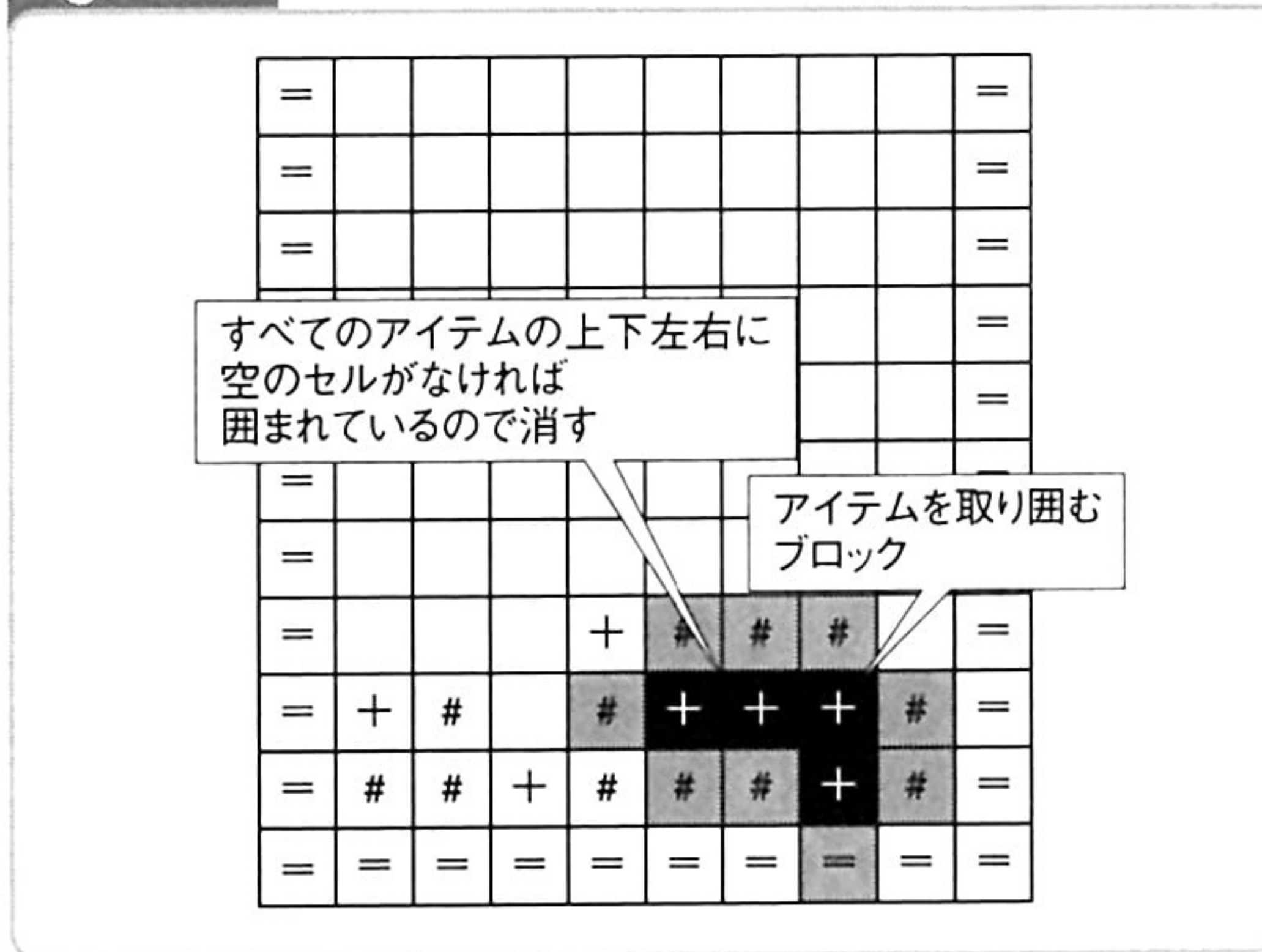



Fig. 4-48 囲まれた複数のアイテムを消す

すべてのアイテムの上下左右に
空のセルがなければ
囲まれているので消す
アイテムを取り囲む
ブロック

イテムの上下左右に空のセルがなければ、それらのアイテムは囲まれているので、消すことができます（Fig. 4-48）。

アイテムが消えたセルは、空のセルにします。空のセルの上にブロックやアイテムがあったら下に落とします。このとき、別のアイテムがブロックに囲まれた場合には、連鎖的に消えます。

なお、本書のサンプルでは、囲んだアイテムは消えますが、ブロックは消えません。ただ、ブロックを消さないと、ステージにどんどんブロックがたまってしまうので、実際のゲームでは消すようにするとよいでしょう。

また、『クレオパトラフォーチュン』のように、囲んだ領域に空間があっても、囲んだという扱いにすることもできます。この場合は「囲んだ領域を塗りつぶす」（→p. 179）の手法を応用するとよいでしょう。ステージを塗りつぶしたときに、塗った領域と塗っていない領域に分かれれば、どちらかが囲まれた領域です（3つ以上の領域に分かれることもあります）。分かれなければ、囲まれた領域はありません。

プログラム

List 4-6はブロックで囲んで消すプログラムです。ステージの移動処理、アイテムの周囲にある空のセルを数える処理、隣接したアイテムを消す処理を掲載しました。

移動処理（Move関数）は、初期状態・入力状態・消去判定状態・消去状態・落下状態に分かれています。これは「エサのブロックを消す」（→p. 227）と同じ構成です。

初期状態では、新しいブロックまたはアイテムを出現させ、入力状態に移行します。入力状態では、レバー入力に応じて、ブロックやアイテムを左右に動かします。また、時間とともにブロックやアイテムを少しずつ落下させます。ブロックやアイテムが着地したら、消去判定状態に移行します。

消去判定状態では、アイテムのセルを探し、アイテムの周囲にある空のセルを数える処理（Count関数）を呼び出します。アイテムの隣に他のアイテムがある場合には、処理を再帰的に

行い、隣接するすべてのアイテムの周囲に、空のセルがあるかどうかを調べます。
空のセルがない場合には、アイテムが囲まれていると判定し、アイテムを消えるアイテムに変化させます。そして、消去状態に移行します。
消去状態では、一定時間が経過するのを待ってから、消えるアイテムを空のセルにして、完全に消します。アイテムが完全に消えるまでには、タイマーを使って、アイテムがだんだん薄くなって消えていくような表示を行います。具体的な処理はステージの描画処理（CSurroundingBlockStageクラスのDraw関数）を参照してください。アイテムを消したら、落下状態に移行します。
落下状態では、空になったセルの上にあるセルを、1段ずつ落下させます。そして、アイテムが連鎖的に消える場合の処理を行うために、消去判定状態に戻ります。

List 4-6　ブロックで囲んで消す（CSurroundingBlockStageクラス）

```
// ステージの移動処理
bool CSurroundingBlockStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 初期状態
    if (State==0) {

        // 新しいブロックの座標
        CX=xs/2;
        CY=0;

        // 新しいブロックの種類（ブロックまたはアイテム）を
        // ランダムに決める
        Type=(Rand.Int31()%2)?'#':'+';

        // タイマーなどを初期化して、
        // 入力状態に移行する
        DropTime=0;
        PrevDown=true;
        State=1;
    }

    // 入力状態
    if (State==1) {

        // レバー入力に応じて、新しいブロックを左右に動かす
        if (!PrevLever) {
            if (is->Left && Cell->Get(CX-1, CY)==' ') CX--;
            if (is->Right && Cell->Get(CX+1, CY)==' ') CX++;
        }
        PrevLever=is->Left||is->Right;
```

```
    // 一定時間ごとに、新しいブロックを落下させる
    DropTime++;
    if (DropTime==60 || (!PrevDown && is->Down)) {
        DropTime=0;

        // 下のセルが空ならば、落下させる
        if (Cell->Get(CX, CY+1)==' ') {
            CY++;
        } else

        // 下のセルが空でなければ、
        // ブロックを固定し、消去判定状態に移行する
        {
            Cell->Set(CX, CY, Type);
            State=2;
        }
    }
    if (!is->Down) PrevDown=false;
}

// 消去判定状態
if (State==2) {

    // タイマーの初期化
    Time=0;

    // ブロックが消えなかったときには、
    // 初期状態に移行する
    State=0;

    // アイテムのセルを探す
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {

            // アイテムのセルが見つかったら、
            // 周囲にある空のセルを数える
            if (
                Cell->Get(x, y)=='+' &&
                Count(x, y)==0
            ) {
                // 空のセルがなければ、
                // アイテムを消えるアイテムにして、
                // 消去状態に移行する
                Erase(x, y);
                State=3;
            }
        }
    }

    // カウントずみのマークを解除する
```

```
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {
            Cell->Set(x, y, Cell->Get(x, y)&0x7f);
        }
    }
}

// 消去状態
if (State==3) {

    // 一定時間が経過したら、消えるアイテムを完全に消す
    Time++;
    if (Time==30) {

        // 消えるアイテムを探す
        for (int y=0; y<ys; y++) {
            for (int x=0; x<xs; x++) {

                // 消えるアイテムを見つけたら、
                // セルを空にする
                if (Cell->Get(x, y)&0x40) {
                    Cell->Set(x, y, ' ');
                }
            }
        }

        // 落下状態に移行する
        Time=0;
        State=4;
    }
}

// 落下状態
if (State==4) {

    // 一定時間が経過したら、下に何もないセルを落下させる
    Time++;
    if (Time==30) {

        // 空のセルを探す
        for (int x=0; x<xs; x++) {
            for (int i=0; i<ys; i++) {

                // 空のセルを見つけたら、
                // 上にあるセルを1段ずつ落下させる
                if (Cell->Get(x, i)==' ') {
                    for (int y=i; y>0; y--) {
                        Cell->Set(x, y, Cell->Get(x, y-1));
                    }
```

```
                    //最上段のセルは空にする
                    Cell->Set(x, 0, ' ');
                }
            }
        }

        //消去判定状態に移行する
        State=2;
    }
}

return true;
}

//アイテムの周囲にある空のセルを数える処理
int CSurroundingBlockStage::Count(int x, int y) {

    //現在位置のセルを取得する
    char c=Cell->Get(x, y);

    //セルがアイテムのときの処理
    if (c=='+') {

        //アイテムをカウントずみとしてマークする
        Cell->Set(x, y, c|0x80);

        //上下左右のセルについても再帰的に調べる
        return
            Count(x-1, y)+
            Count(x+1, y)+
            Count(x, y-1)+
            Count(x, y+1);
    } else

    //セルが空ならば、
    //空のセルの個数(1)を返す
    if (c==' ') {
        return 1;
    } else

    //それ以外の場合(セルがブロックまたは壁の場合)には、
    //空のセルの個数(0)を返す
    {
        return 0;
    }
}

//隣接したアイテムを消す処理
void CSurroundingBlockStage::Erase(int x, int y) {
```

```
    //現在位置のセルを取得する
    char c=Cell->Get(x, y);

    //セルがアイテムのときの処理
    if (c==(char)('+'|0x80)) {

        //アイテムを消えるアイテムとしてマークする
        Cell->Set(x, y, c|0x40);

        //上下左右のセルについても、
        //アイテムを消す処理を再帰的に行う
        Erase(x-1, y);
        Erase(x+1, y);
        Erase(x, y-1);
        Erase(x, y+1);
    }
}
```

SAMPLE

「SURROUNDING BLOCK」は「ブロックで囲んで消す」のサンプルです。

ステージ上方からブロックまたはアイテムが降ってきます。ブロックやアイテムはレバーの左右（カーソルキーの左右）で左右に移動します。レバーの下（カーソルキーの下）を入力すると、落下スピードが上がります。

ブロックやアイテムが着地したときに、ブロックに囲まれたアイテムがあると、消すことができます。アイテムを消すと、上にあったブロックやアイテムが落ちてきます。このとき、再びブロックに囲まれたアイテムがあると、連鎖的に消えます。

SURROUNDING BLOCK → p. 388

つながったブロックを消す

縦横に隣接した同じ種類のブロックを消すアクションです。これは落ち物パズルゲームではなく、ステージに配置されたブロックを選択して消す形式になっています。

ステージには多数のブロックが配置されています（Fig. 4-49）。ブロックには種類があり、色や形が異なります。本書のサンプルでは、ブロックの種類ごとに形を変えています。

ステージにはカーソルが表示されています。カーソルはレバー入力で上下左右に動かすことができます。

ボタンを押すと、カーソルの位置にあるブロックを消すことができます（Fig. 4-50）。ただし、同じ種類のブロックが2個以上、上下左右に隣接していないと、消すことはできません。2個以

Fig. 4-49 配置されたブロック

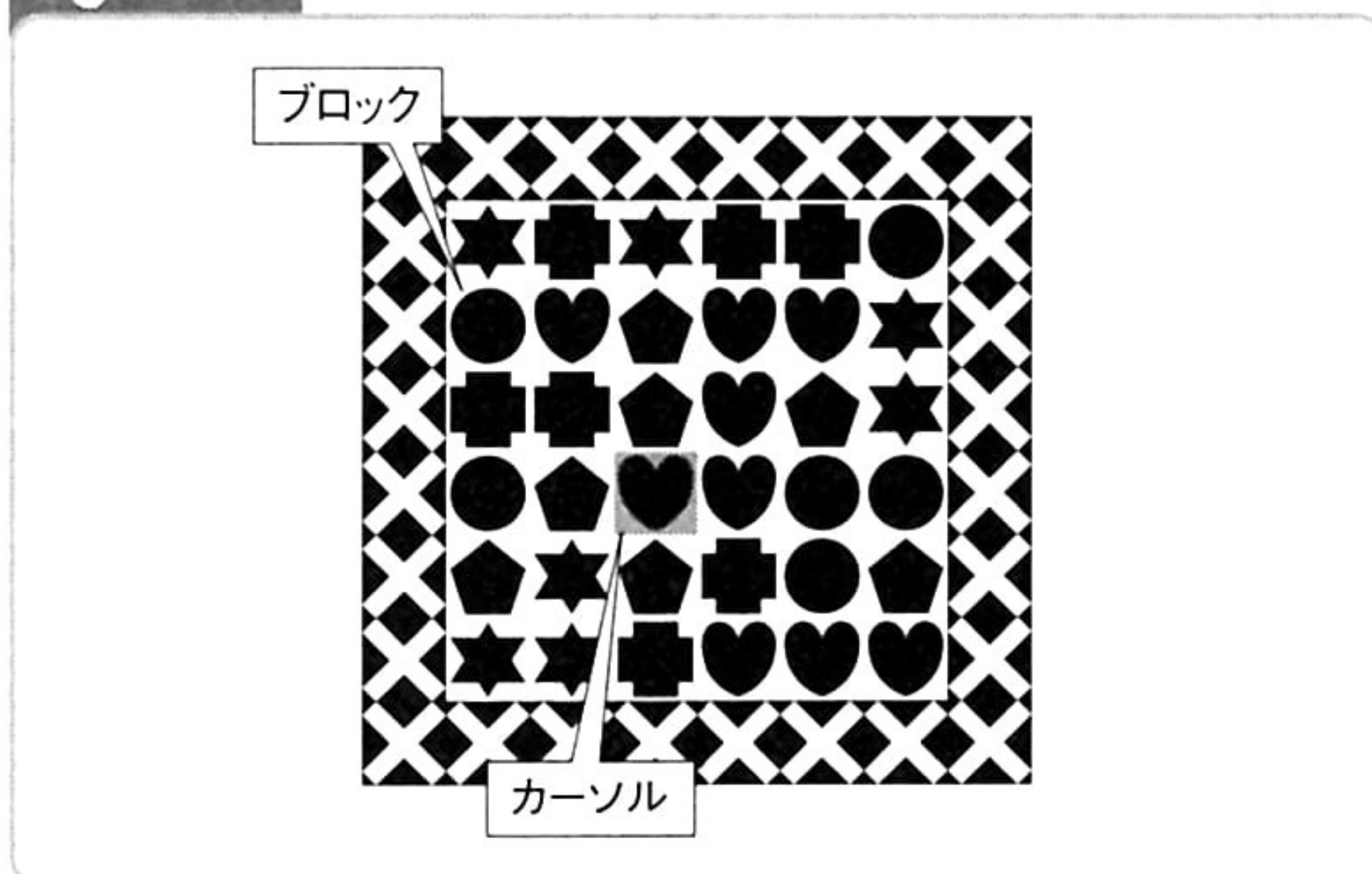



Fig. 4-50 ブロックを消す

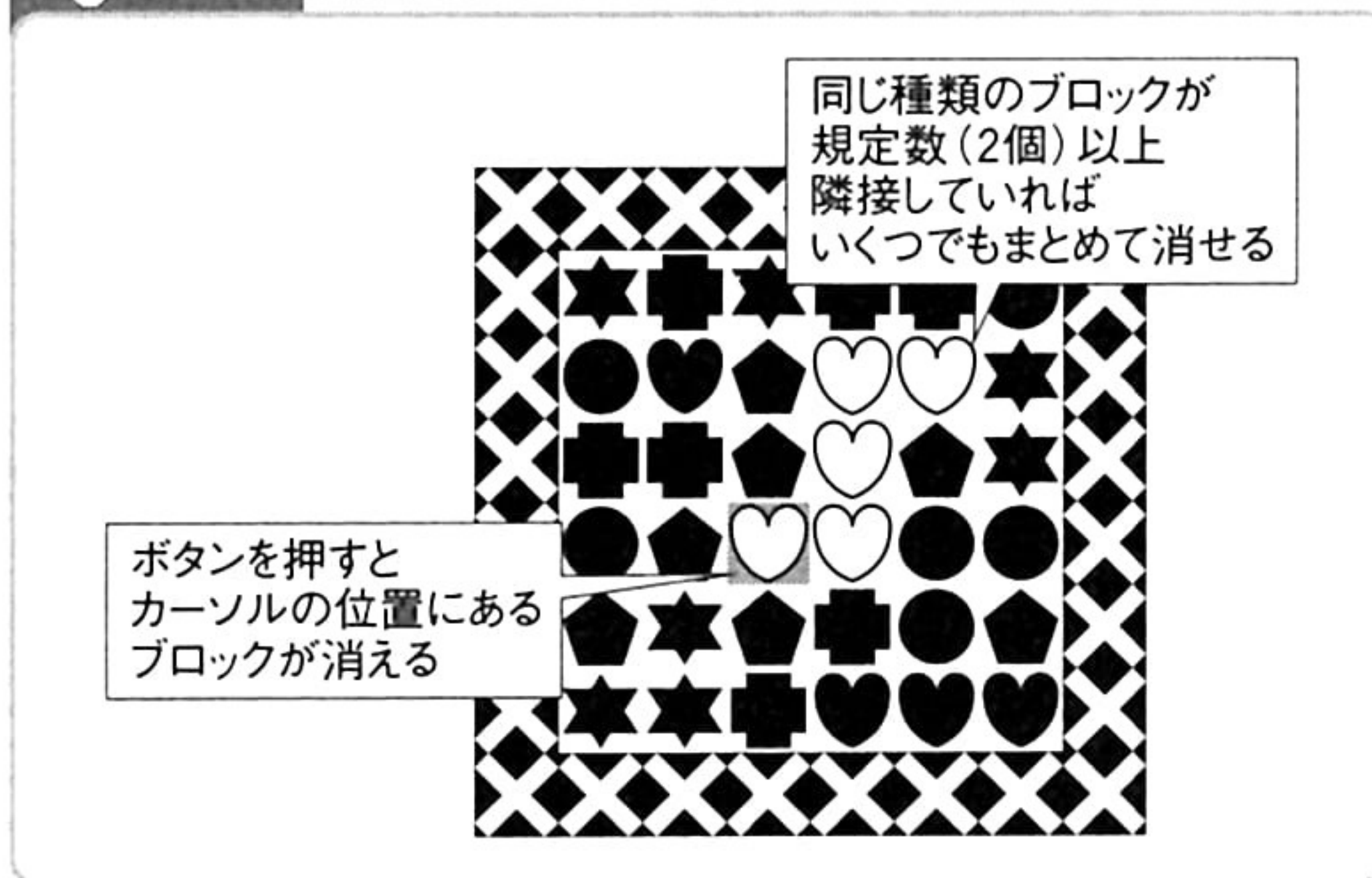



Fig. 4-51 上にあったブロックが落ちる

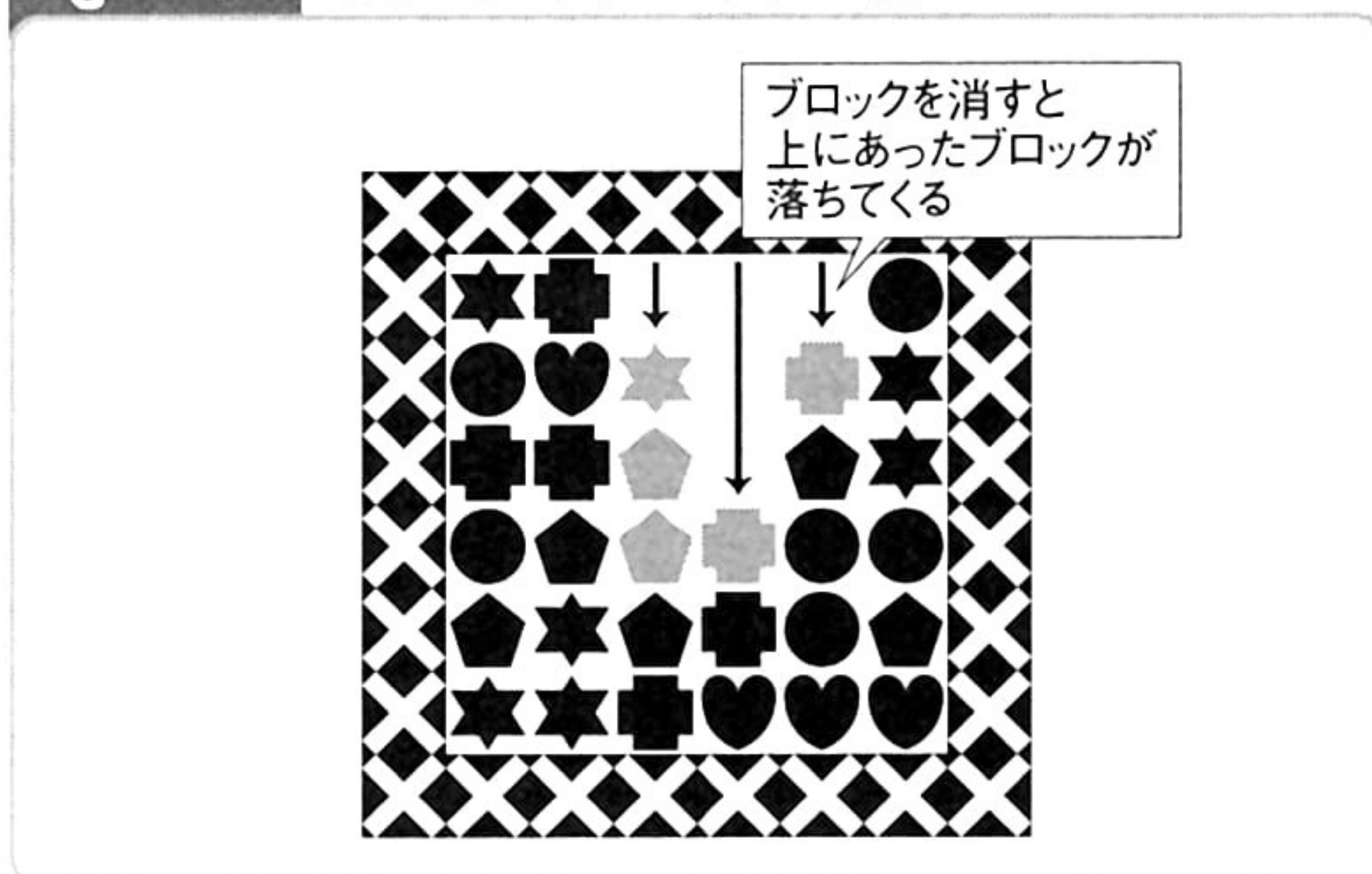
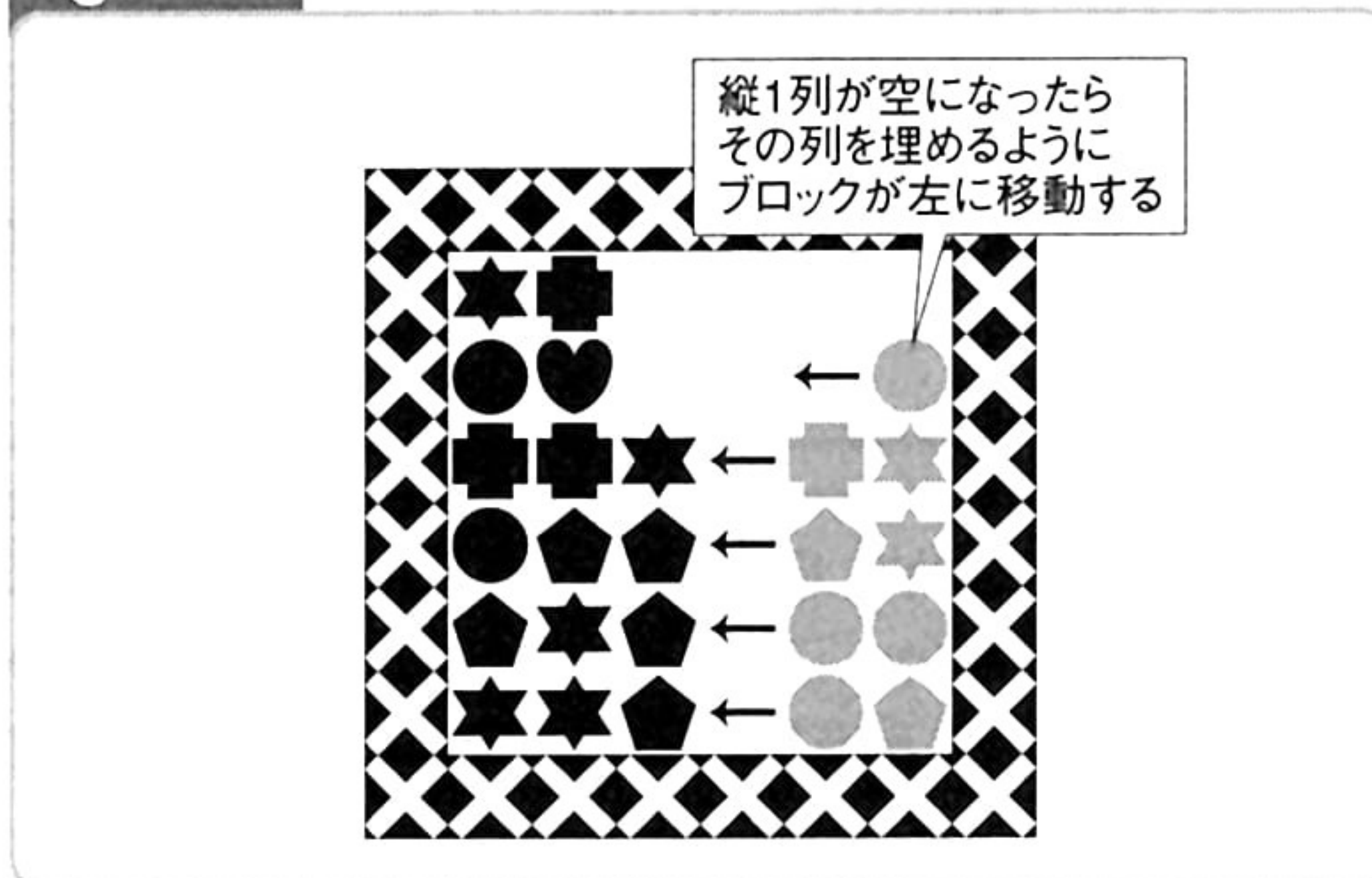



Fig. 4-52 ブロックが左に移動する

上隣接している場合には、いくつのブロックが隣接していても、まとめて消すことができます。

ブロックを消すと、上にあったブロックが落ちてきます（Fig. 4-51）。また、ブロックを消した結果、縦一列が空になった場合には、その列よりも右にあるブロックが、列を埋めるように左に移動します（Fig. 4-52）。

つながったブロックを消すアクションを使ったゲームには『さめがめ』があります。このゲームでは、できるだけ多くのブロックをまとめて消すことによって、高いスコアが得られます。ブロックの最初の配置や、ブロックを消す順番によって、大きくスコアが変わってくるのが面白いところです。

アルゴリズム

つながったブロックを消すアクションは、落ち物パズルゲームではありませんが、「連鎖的に消す」（→p. 106）と似た方法で実現できます。ブロックを落とすかわりに、カーソルを使って消すブロックを選択します。

まず、ステージをセルで表現します（Fig. 4-53）。ステージの壁は「＝」で、ブロックは「0〜4」の数字で表しました。

ボタンを押したら、カーソルの位置にあるブロックについて、同じ種類のブロックが隣接している数を調べます（Fig. 4-54）。上下左右に同じ種類のブロックがあったら、そのブロックのさらに上下左右についても再帰的に調べます（Fig. 4-55）。

同じ種類のブロックが規定数（ここでは2個）以上隣接していたら、隣接したブロックをすべて消します（Fig. 4-56）。消したブロックのセルは空にします。

空になったセルの上のセルは、下に落とします（Fig. 4-57）。また、縦1列が空になった場合には、空になった列よりも右にあるセルを、左に1列ずつ移動します（Fig. 4-58）。

Fig. 4-53　ステージをセルで表現する

=	=	=	=	=	=	=	=
=	1	3	1	3	3	0	=
=	0	2	4	2	2	1	=
=	3	3	4	2	4	1	=
=	0	4	2	2	0	0	=
=	4	1	4	3	0	4	=
=	1	1	3	2	2	2	=
=	=	=	=	=	=	=	=

Fig. 4-54　隣接するブロックの数を調べる

Fig. 4-55　隣接するブロックの上下左右を再帰的に調べる

Fig. 4-56　隣接したブロックを消す

Fig. 4-57　空になったセルの上のセルを落とす

Fig. 4-58　空になった列の右のセルを移動する

プログラム

List 4-7はつながったブロックを消すプログラムです。ステージの移動処理と、ブロックを消す処理を掲載しました。

移動処理(Move関数)は、入力状態・消去状態・横移動状態に分かれています。入力状態では、レバー入力に応じて、カーソルを上下左右に動かします。ボタンを押したら、カーソルの位置にあるブロックを消すために、ブロックを消す処理(Erase関数)を呼び出します。

ブロックを消す処理は、隣接した同じ種類のブロックを数える処理を兼ねています。上下左右に同じ種類のセルがあるかどうかを調べて、セルがある場合には、そのセルについても上下左右のセルを調べます。同じ種類のセルが隣接しているかぎり、再帰的に処理を行うことによって、隣接したすべてのブロックにマークを付けます。

規定数(2個)以上のブロックが隣接していたら、消去状態に移行します。消去状態では、一定時間が経過するのを待ってから、マークを付けたブロックを消去します。そして、消去したブロックの上にあるセルを落下させます。

消去状態の次は、横移動状態に移行します。横移動状態では、一定時間が経過するのを待ってから、縦1列が空になった列を探します。空の列があったら、その列よりも右にあるセルを、1列ずつ左に移動します。

List 4-7 つながったブロックを消す(CConnectedBlockStageクラス)

```
// ステージの移動処理
bool CConnectedBlockStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 入力状態
    if (State==0) {

        // レバー入力に応じて、カーソルを上下左右に動かす
        if (!PrevLever) {
            if (is->Left && CX>1) CX--; else
            if (is->Right && CX<xs-2) CX++; else
            if (is->Up && CY>2) CY--; else
            if (is->Down && CY<ys-2) CY++;
        }
        PrevLever=is->Left||is->Right||is->Up||is->Down;

        // ボタンを押したら、カーソルの位置にあるブロックを消す
        if (!PrevButton && is->Button[0]) {

            // カーソルの位置にあるセルを取得する
            char c=Cell->Get(CX, CY);
```

```
            // セルが空ではなく、
            // 同じ種類のセルが規定数（2個）以上隣接していたら、
            // ブロックを消す
            if (
                c!=' ' &&
                Erase(CX, CY, c, 0x80)>=
                    CONNECTED_BLOCK_ERASE
            ) {
                // ブロックを消す
                Erase(CX, CY, c|0x80, 0x40);

                // 消去状態に移行する
                Time=0;
                State=1;
            }
        }
        PrevButton=is->Button[0];

        // セルに付けたカウントずみのマークを解除する
        for (int y=2; y<ys-1; y++) {
            for (int x=1; x<xs-1; x++) {
                Cell->Set(x, y, Cell->Get(x, y)&0x7f);
            }
        }
    }

    // 消去状態
    if (State==1) {

        // 一定時間が経過したら、消えるブロックを完全に消す
        Time++;
        if (Time==30) {

            // 消えるブロックを探す
            for (int x=1; x<xs-1; x++) {
                for (int y=2; y<ys-1; y++) {

                    // 消えるブロックを見つけたときの処理
                    if (Cell->Get(x, y)&0x40) {

                        // 消えるブロックよりも上にあるセルを
                        // 1段ずつ落下させる
                        for (int i=y; i>2; i--) {
                            Cell->Set(x, i, Cell->Get(x, i-1));
                        }

                        // 最上段のセルは空にする
                        Cell->Set(x, 2, ' ');
                    }
```

```
                }
            }

            // 横移動状態に移行する
            Time=0;
            State=2;
        }
    }

    // 横移動状態
    if (State==2) {

        // 一定時間が経過したら、
        // 空いた列よりも右のセルを左に移動する
        Time++;
        if (Time==30) {

            // 空いた列を探す
            for (int x=xs-2; x>=1; x--) {

                // 上端から下端まで
                // すべてのセルが空かどうかを調べる
                int y;
                for (y=2; y<ys-1 && Cell->Get(x, y)==' '; y++) ;

                // すべてのセルが空の場合の処理
                if (y==ys-1) {

                    // 空の列よりも右のセルを、
                    // 1列ずつ左に移動する
                    for (int i=x; i<xs-2; i++) {
                        for (int j=2; j<ys-1; j++) {
                            Cell->Set(i, j, Cell->Get(i+1, j));
                        }
                    }

                    // 右端のセルは空にする
                    for (int j=2; j<ys-1; j++) {
                        Cell->Set(xs-2, j, ' ');
                    }
                }
            }

            // 入力状態に移行する
            State=0;
        }
    }

    return true;
}
```

```
// ブロックを消す処理
// 隣接した同じ種類のブロックを数える処理も兼ねている
int CConnectedBlockStage::Erase(int x, int y, char c, int mask) {

    // 現在位置のセルが指定された種類のセルならば、
    // セルをマークする
    if (Cell->Get(x, y)==c) {

        // 指定されたビットをセットすることによって、
        // セルをマークする
        Cell->Set(x, y, c|mask);

        // 上下左右のセルについても再帰的に調べて、
        // 隣接した同じ種類のセルの数を返す
        return
            1+
            Erase(x-1, y, c, mask)+
            Erase(x+1, y, c, mask)+
            Erase(x, y-1, c, mask)+
            Erase(x, y+1, c, mask);
    }

    // 現在位置のセルは指定された種類のセルではないので、
    // 隣接した同じ種類のセルの数としては0を返す
    return 0;
}
```

SAMPLE

「CONNECTED BLOCK」は「つながったブロックを消す」のサンプルです。

レバーの上下左右（カーソルキーの上下左右）でカーソルが動きます。ボタン0（Zキー）を押すと、カーソルの位置にあるブロックを消すことができます。

同じ種類のブロックが2個以上隣接していれば、消すことができます。同じ種類のブロックが上下左右に隣接しているかぎり、いくつでもまとめてブロックを消すことができます。

ブロックを消すと、上にあったブロックが落ちてきます。ブロックを消して、縦一列が空になったときには、その列よりも右にあるブロックが左にスライドします。

CONNECTED BLOCK → p. 388

ブロックを引き寄せて撃つ

引き寄せたブロックを撃つアクションです。引き寄せたり撃ったりを繰り返すことで、ブロックの配置を変化させ、同じ種類のブロックを並べて消します。

ステージにはブロックが積まれています（Fig. 4-59）。左端にいるキャラクターは、レバー入力で上下に動かすことができます。

ボタンを押すと、キャラクターがブロックを引き寄せます（Fig. 4-60）。引き寄せるブロックは、キャラクターと同じ高さにあるブロックのうち、最もキャラクターに近いブロックです。

ブロックを引き寄せたときに、上に他のブロックがあると、そのブロックは落下します（Fig. 4-61）。このとき、同じ種類のブロックが規定数（ここでは3個）以上隣接すると、ブロックを消すことができます（Fig. 4-62）。

Fig. 4-59 ステージの構成

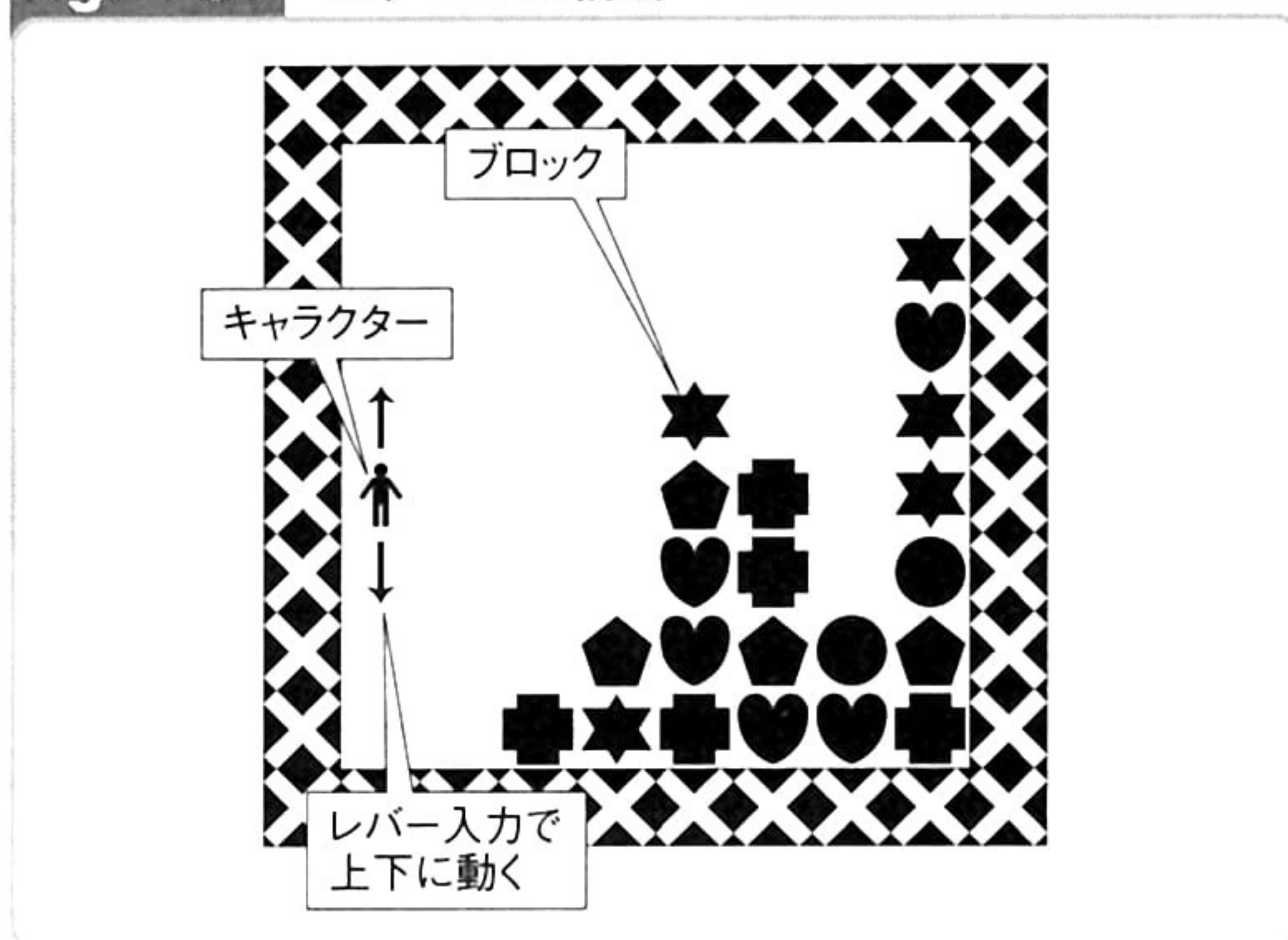



Fig. 4-60 ブロックを引き寄せる

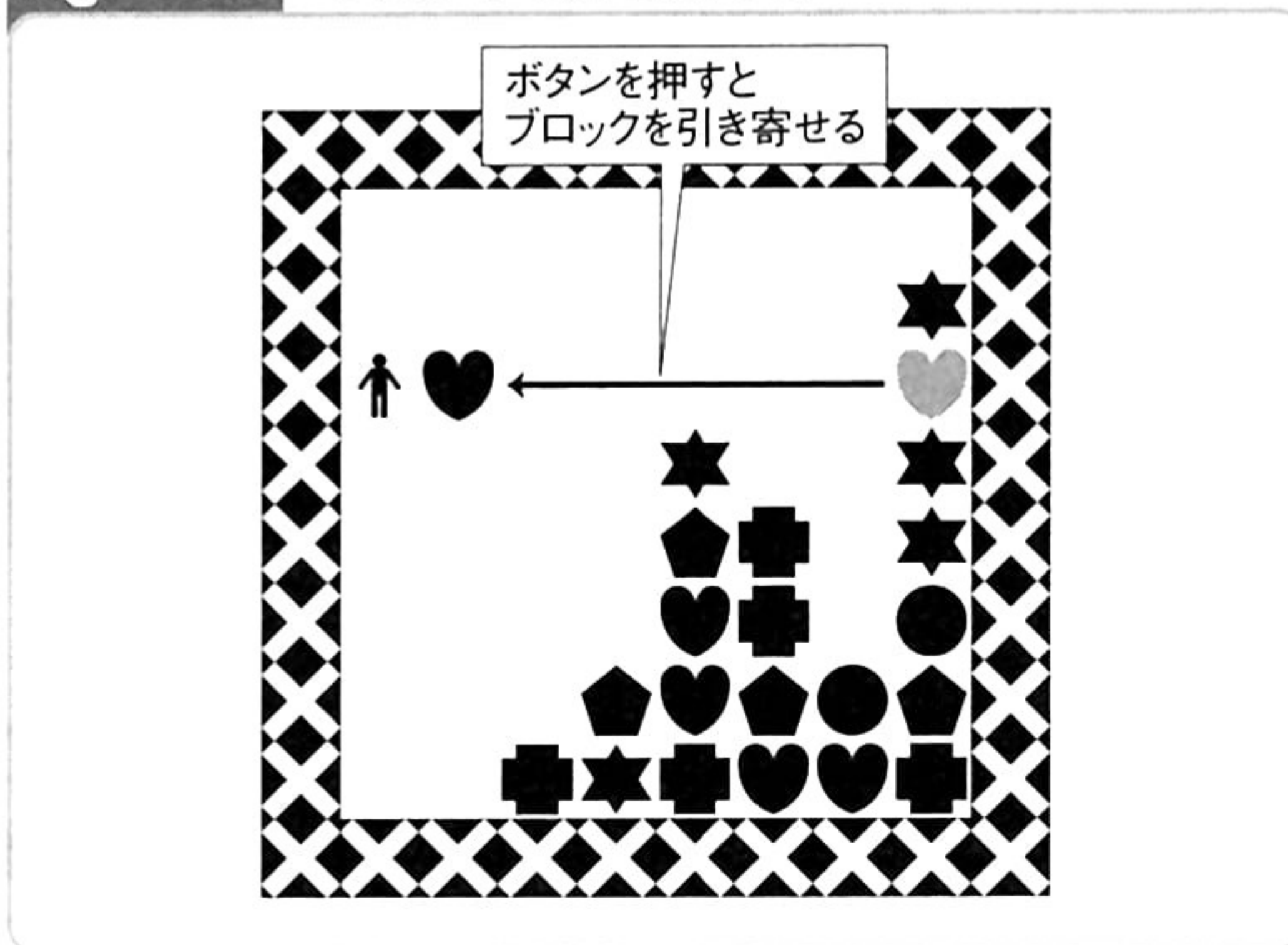



Fig. 4-61 引き寄せた上のブロックが落下する

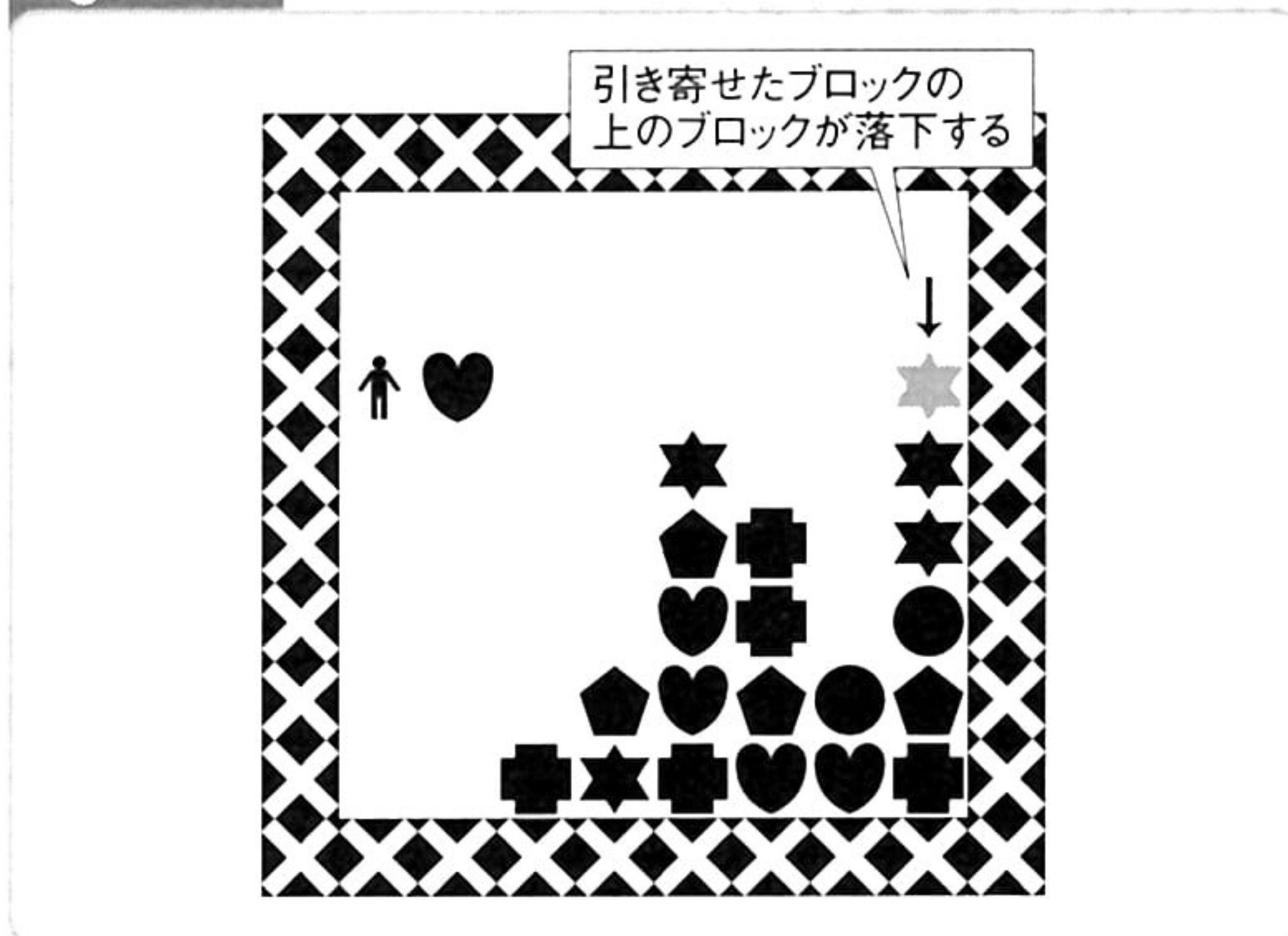
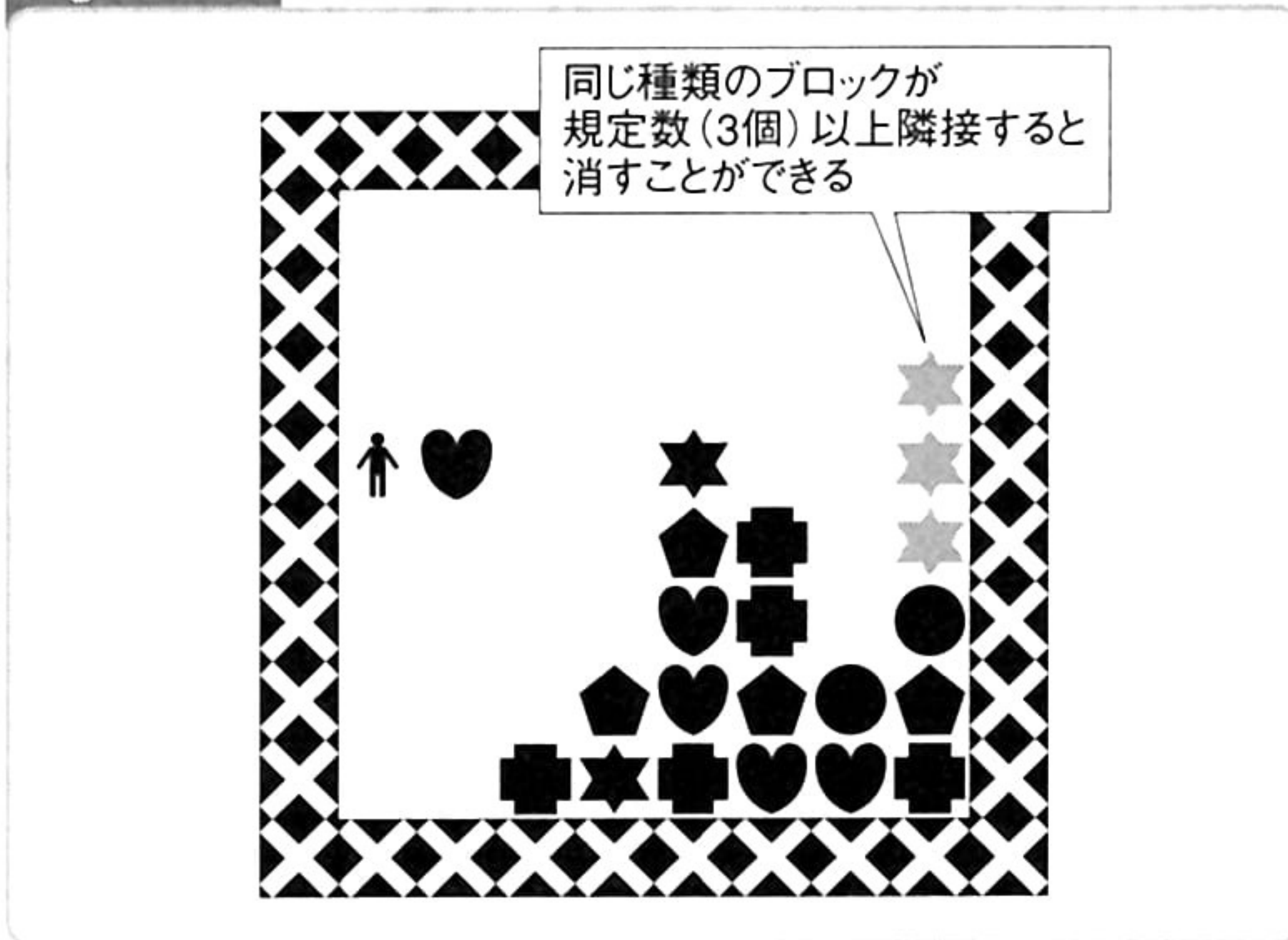



Fig. 4-62 引き寄せを利用してブロックを消す

キャラクターが引き寄せたブロックを持っているときに、ボタンを押すと、ブロックを撃つことができます(Fig. 4-63)。撃ったブロックは、キャラクターと同じ高さで右に飛んでいき、他のブロックに当たると止まります。このとき、下に他のブロックがなければ、撃ったブロックは落下します(Fig. 4-64)。

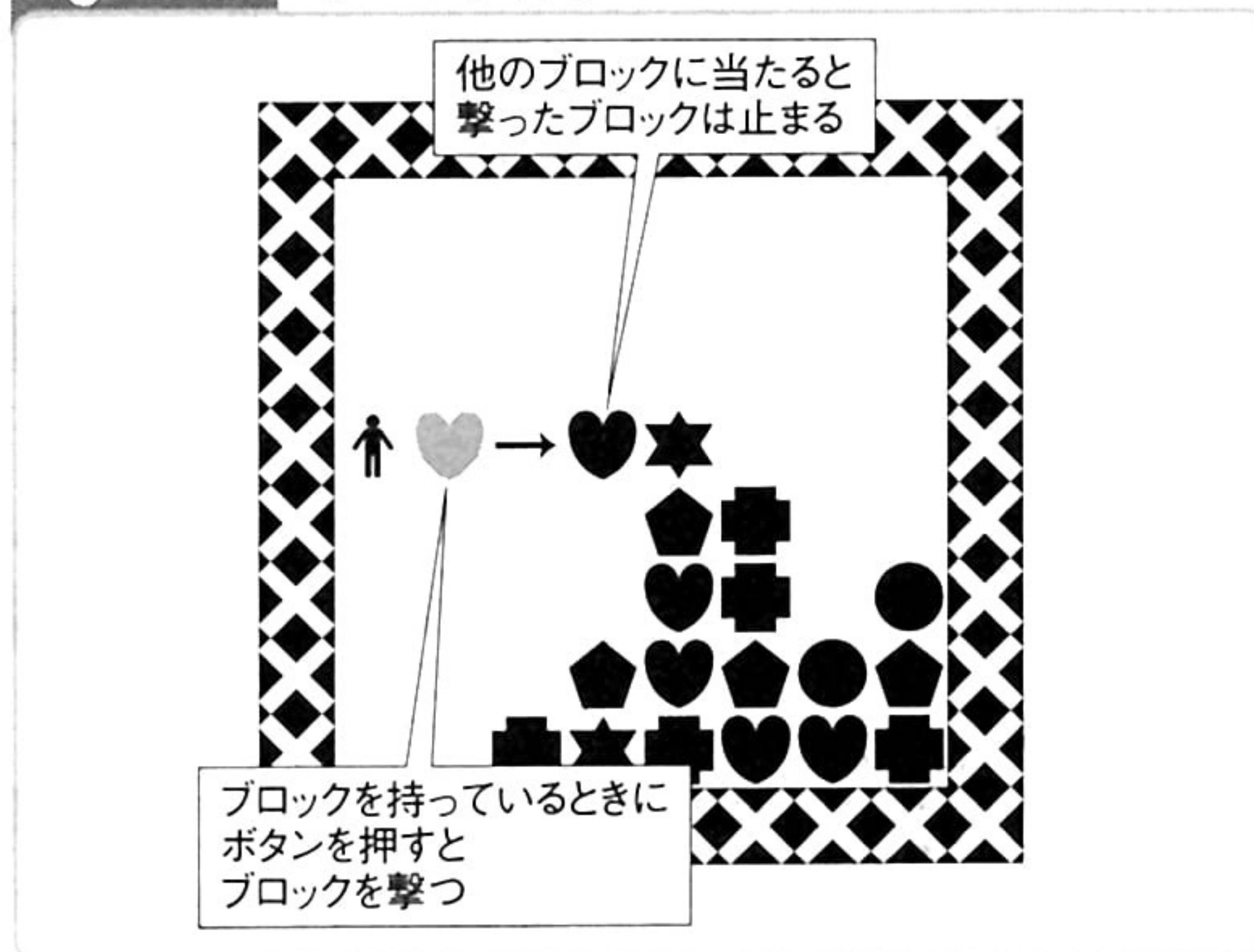
撃ったブロックが着地したときに、同じ種類のブロックが規定数以上並ぶと、ブロックを消すことができます(Fig. 4-65)。消えたブロックの上にあるブロックは落下します(Fig. 4-66)。このとき、再び同じ種類のブロックが規定数以上隣接すると、連鎖的に消えます(Fig. 4-67)。また、縦一列のブロックがすべて消えたときには、その列よりも左にある列が右に移動します(Fig. 4-68)。

ブロックを引き寄せて撃つアクションは『このeたこ』に採用されています。このゲームの特徴は、キャラクターが画面の中央にいて、両側からブロックが迫ってくることです。どちらか片側のブロックが中央に達するとゲームオーバーなので、両側のブロックを素早く消していく必要があります。片側のブロックを引き寄せて、反対側に撃つことができる点もユニークです。

Fig. 4-63 ブロックを撃つ

他のブロックに当たると
撃ったブロックは止まる

ブロックを持っているときに
ボタンを押すと
ブロックを撃つ

Fig. 4-64 撃ったブロックが落ちる

下に他のブロックがなければ
撃ったブロックは落ちる

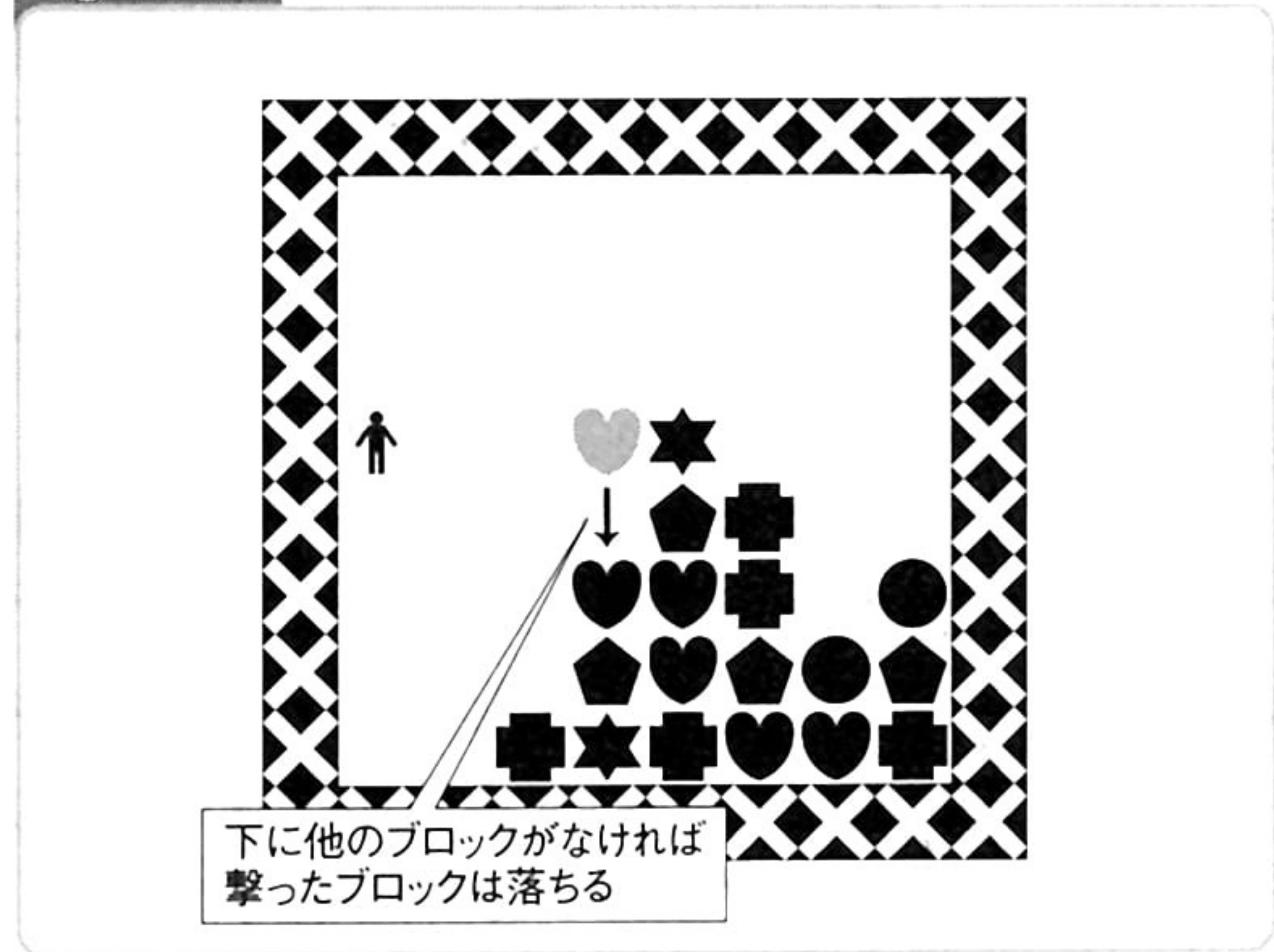



Fig. 4-65 撃ったブロックを利用してブロックを消す

同じ種類のブロックが
規定数(3個)以上隣接すると
消すことができる

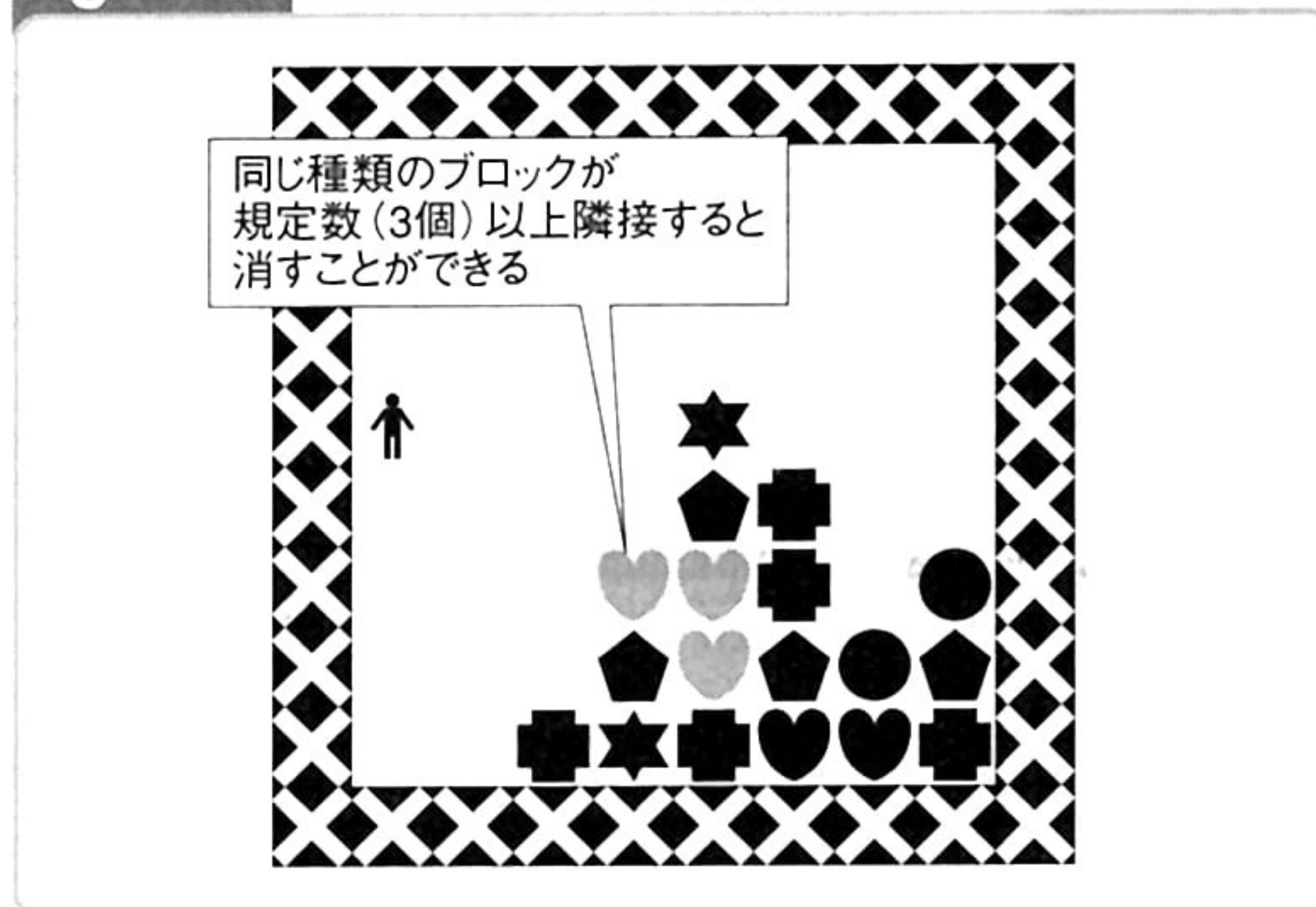
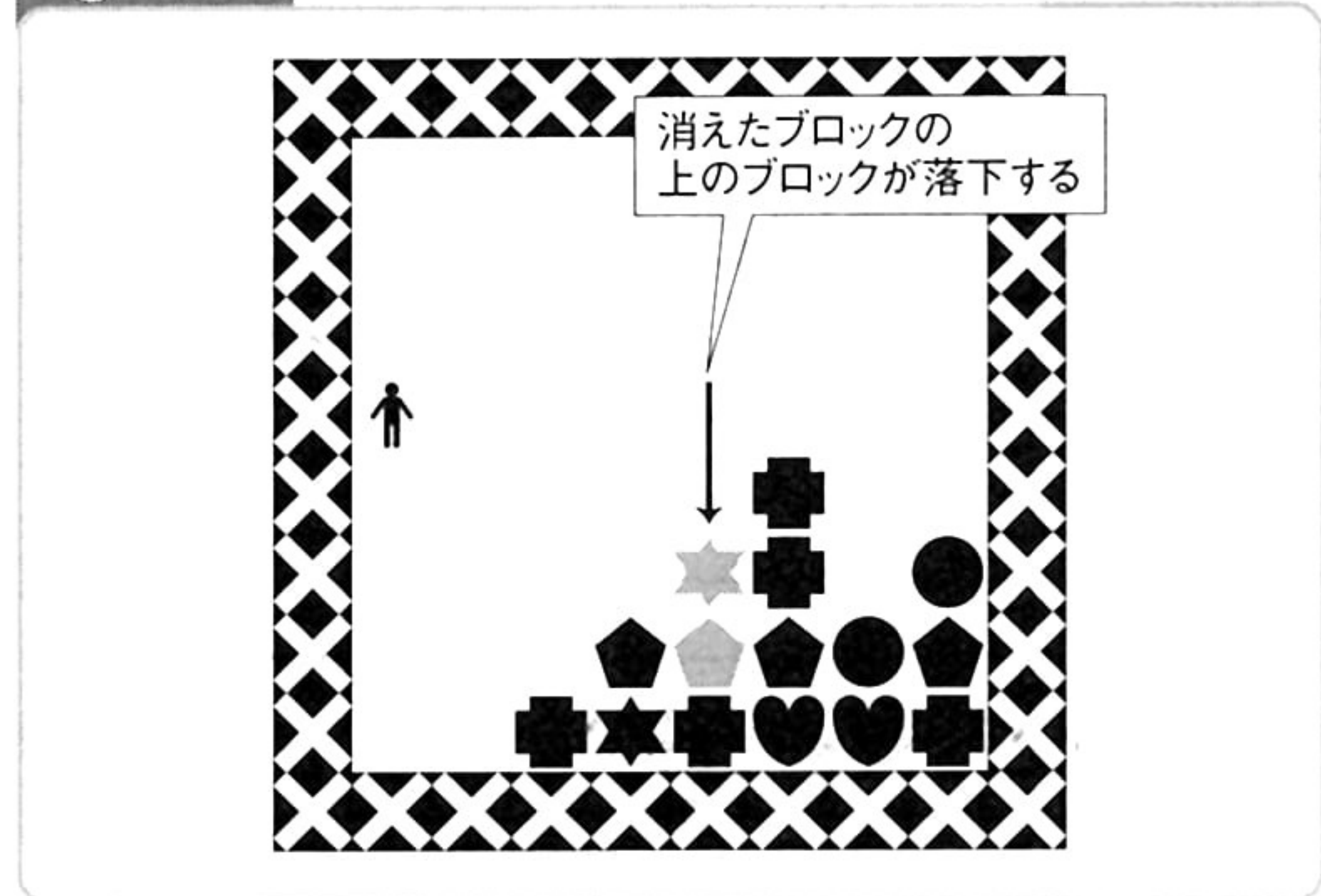



Fig. 4-66 消えたブロックの上のブロックが落下する

消えたブロックの
上のブロックが落下する

Fig. 4-67 連鎖的に消える

再び同じ種類のブロックが
3個以上揃うと
連鎖的に消える

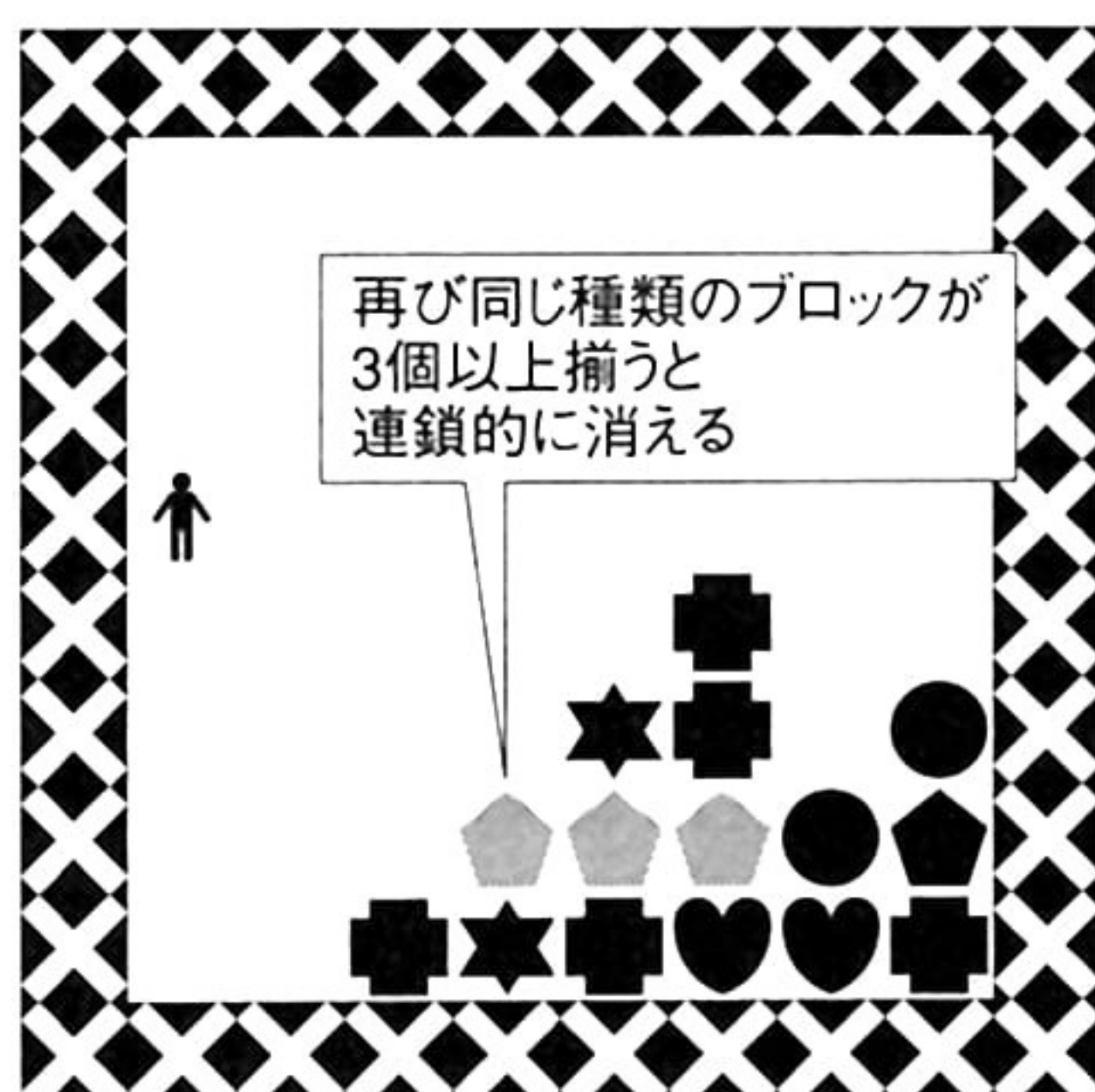
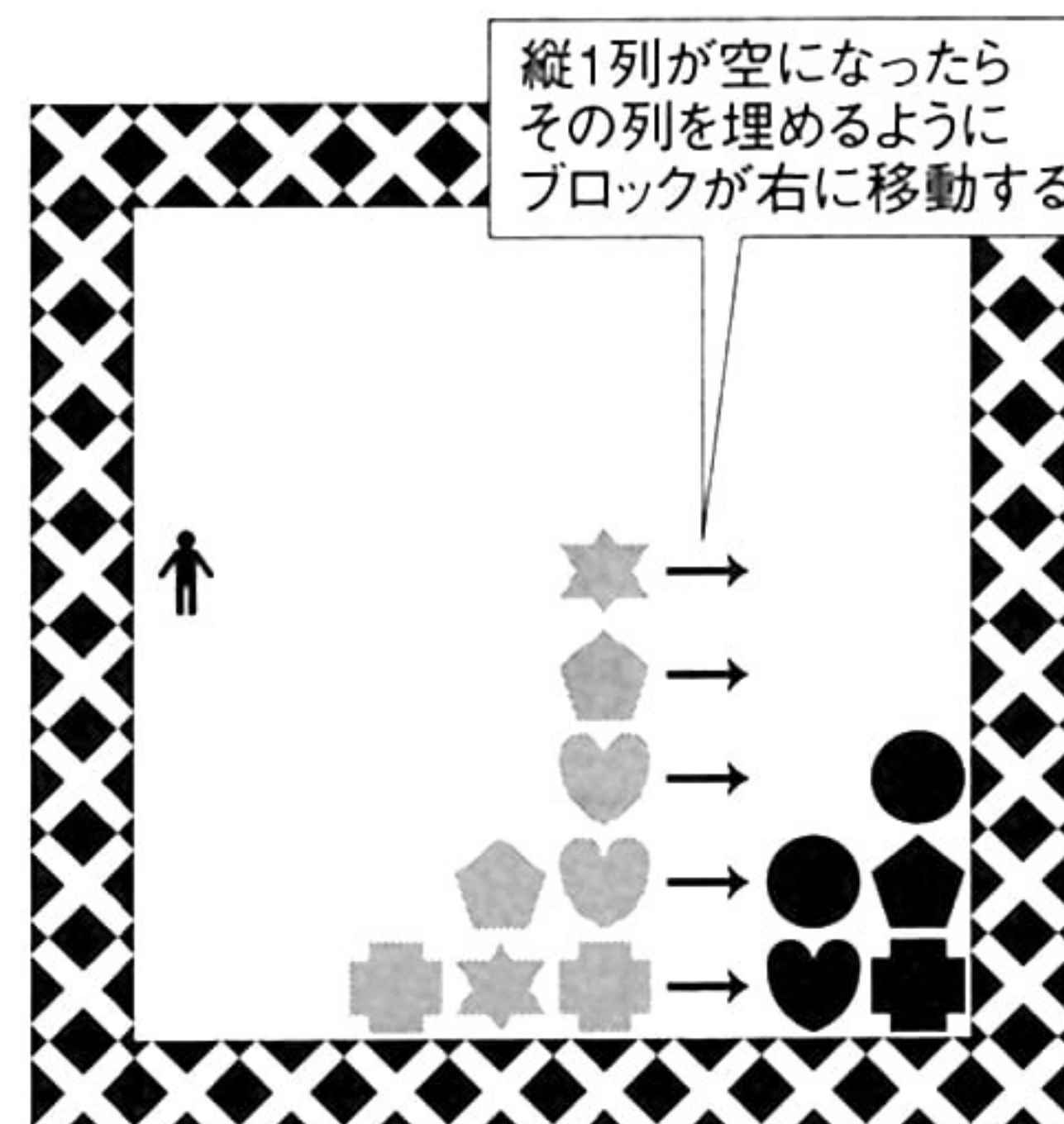



Fig. 4-68 消えた列の左の列が右に移動する

縦1列が空になったら
その列を埋めるように
ブロックが右に移動する

アルゴリズム

ブロックを引き寄せて撃つアクションは、「つながったブロックを消す」(→p. 241)と似た方法で実現できます。カーソルを動かして消すブロックを選択するかわりに、ブロックを引き寄せたり撃ったりします。

まず、ステージをセルで表現します(Fig. 4-69)。ステージの壁は「＝」で、ブロックは「0〜4」の数字で表しました。

ボタンを押したら、キャラクターの右に向かって、ブロックのセルを探します(Fig. 4-70)。ブロックが見つかったら、そのセルを空にして、画面にはブロックが引き寄せられる様子を描画します。また、引き寄せたブロックの種類を記録しておきます。

ブロックを持っているときにボタンを押したら、キャラクターの右に向かってブロックを撃ちます。画面にはブロックが飛ぶ様子を描画しながら、右に向かってセルを調べて、他のブロックや壁にぶつかったら、そこにブロックを固定します(Fig. 4-71)。

Fig. 4-69 ステージをセルで表現する

壁(＝)

＝	＝	＝	＝	＝	＝	＝	＝	＝	＝
＝									＝
＝								1	＝
＝								2	＝
＝					1			1	＝
＝					4	3		1	＝
＝					2	3		0	＝
＝				4	2	4	0	4	＝
＝			3	1	3	2	2	3	＝
＝	＝	＝	＝	＝	＝	＝	＝	＝	＝

ブロック(0〜4)

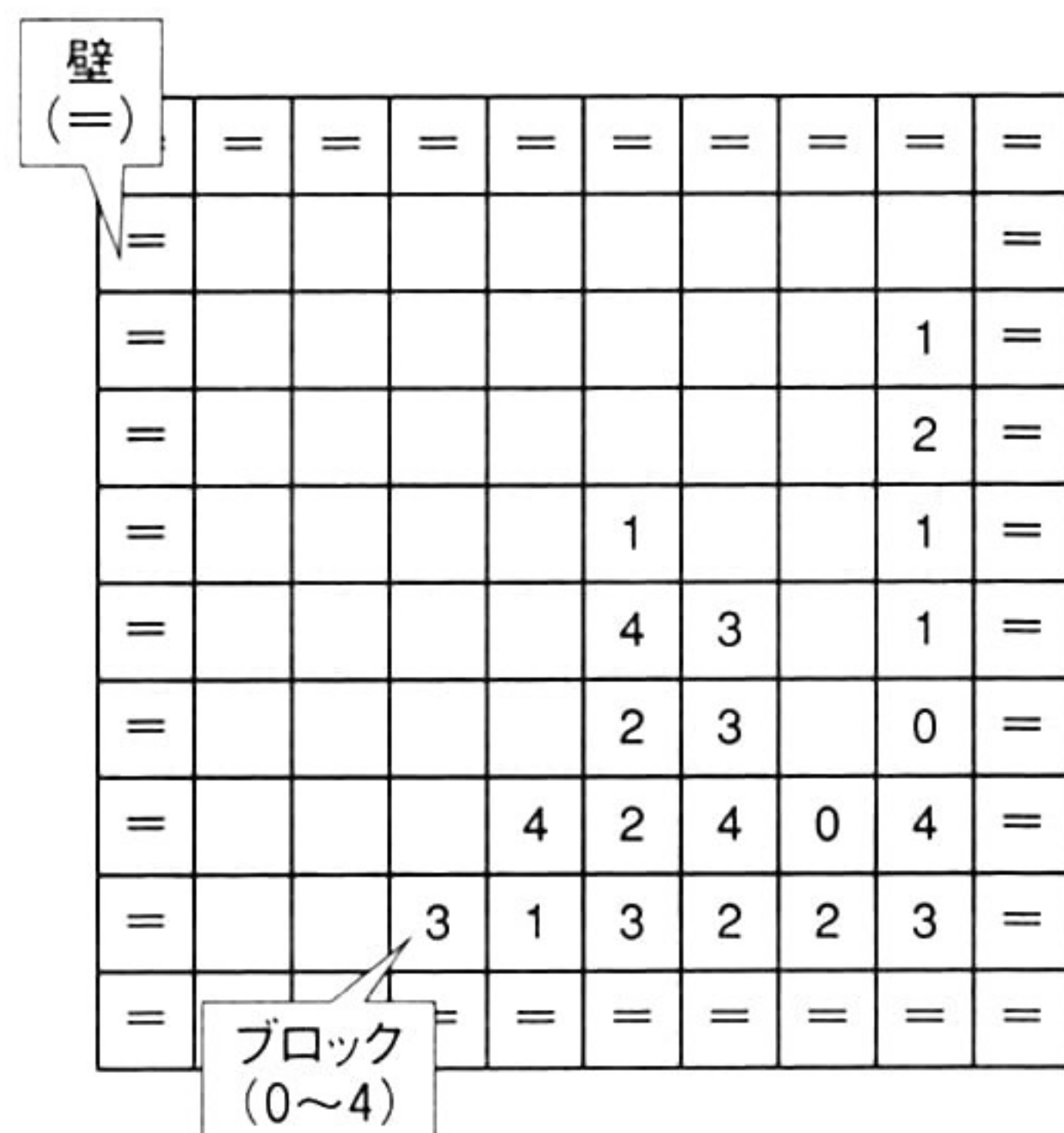



Fig. 4-70 引き寄せるブロックのセルを探す

キャラクターの右に向かって
ブロックのセルを探す

キャラクター

引き寄せたブロックの
種類を記録する

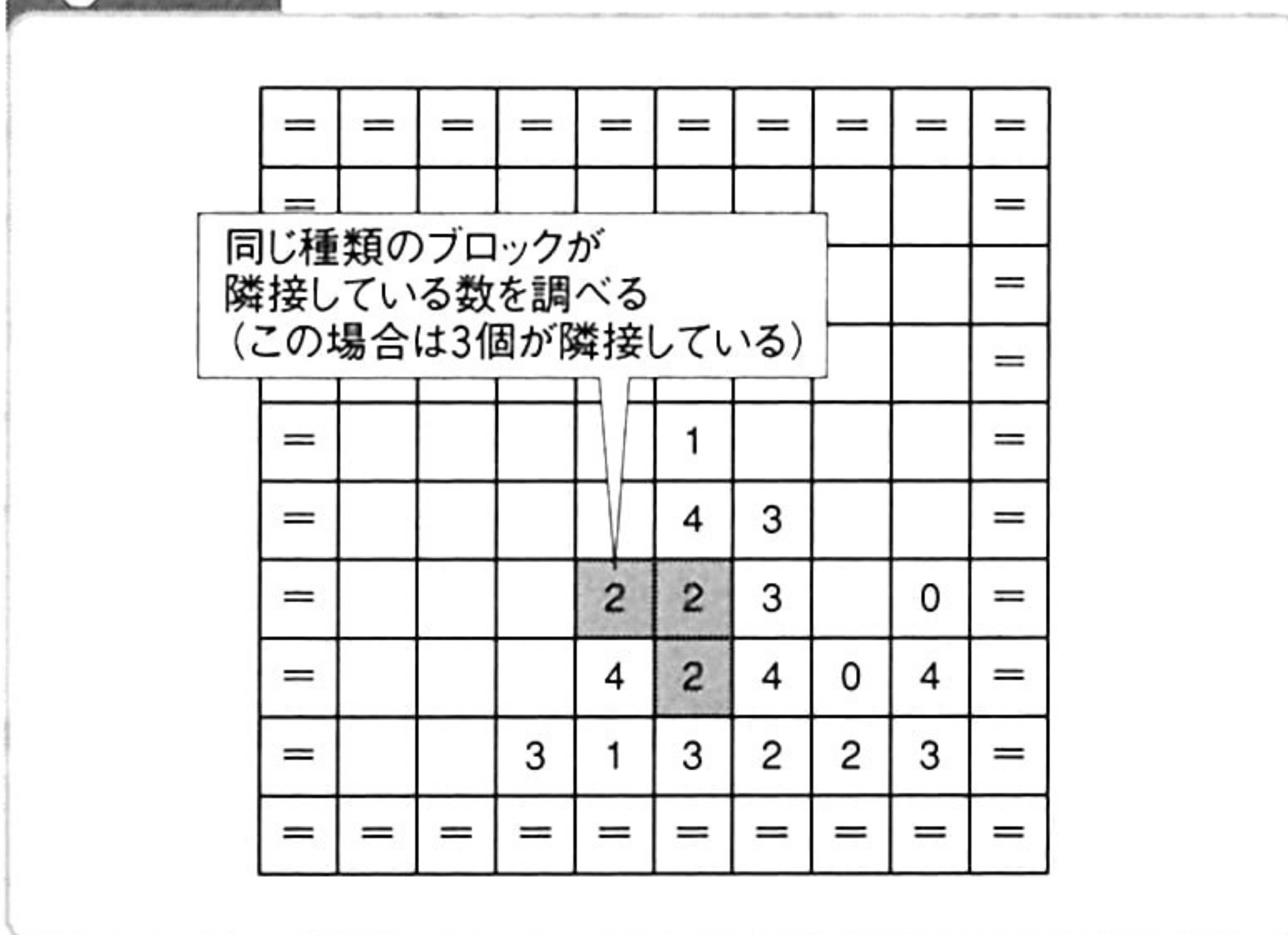
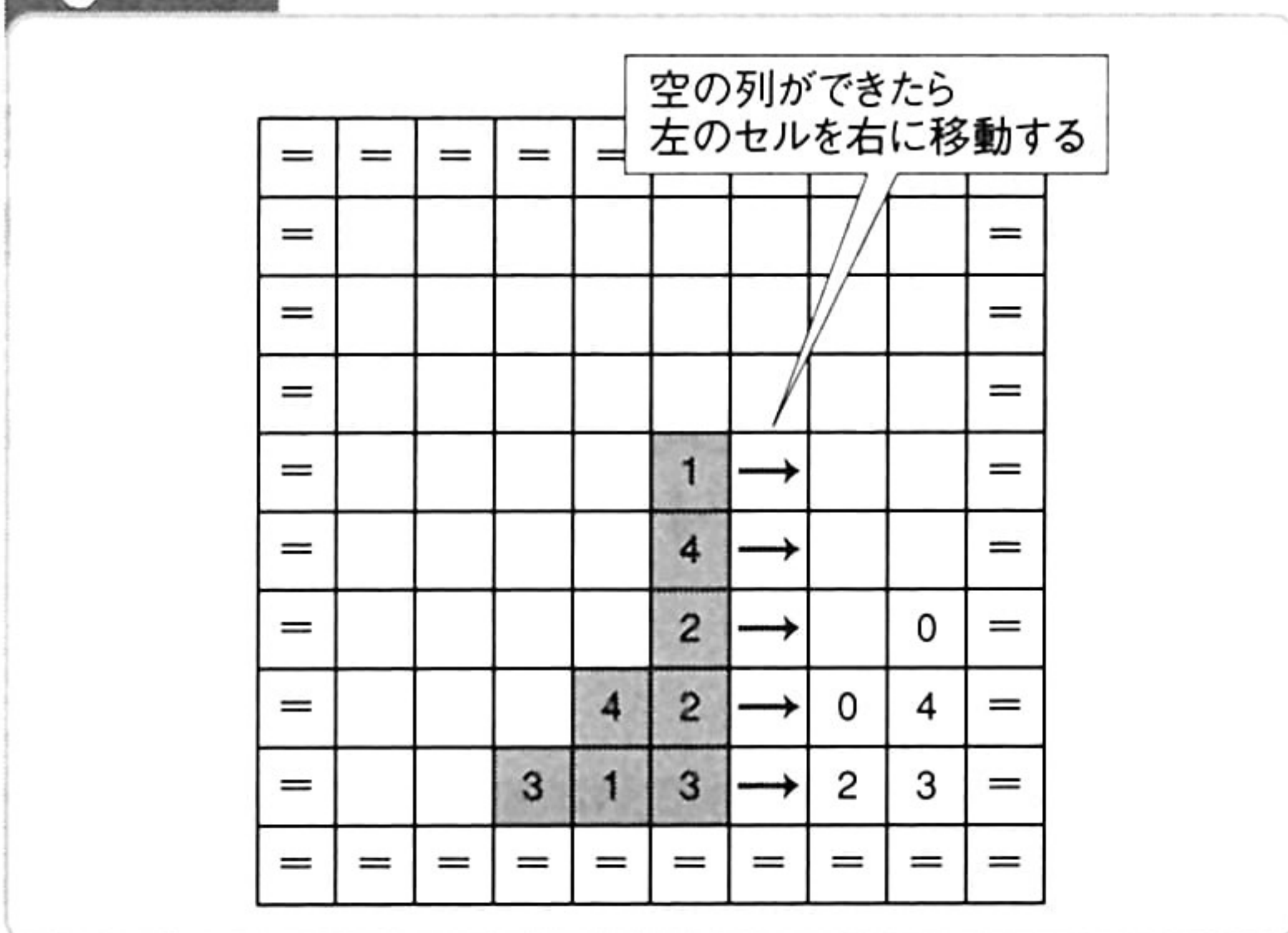
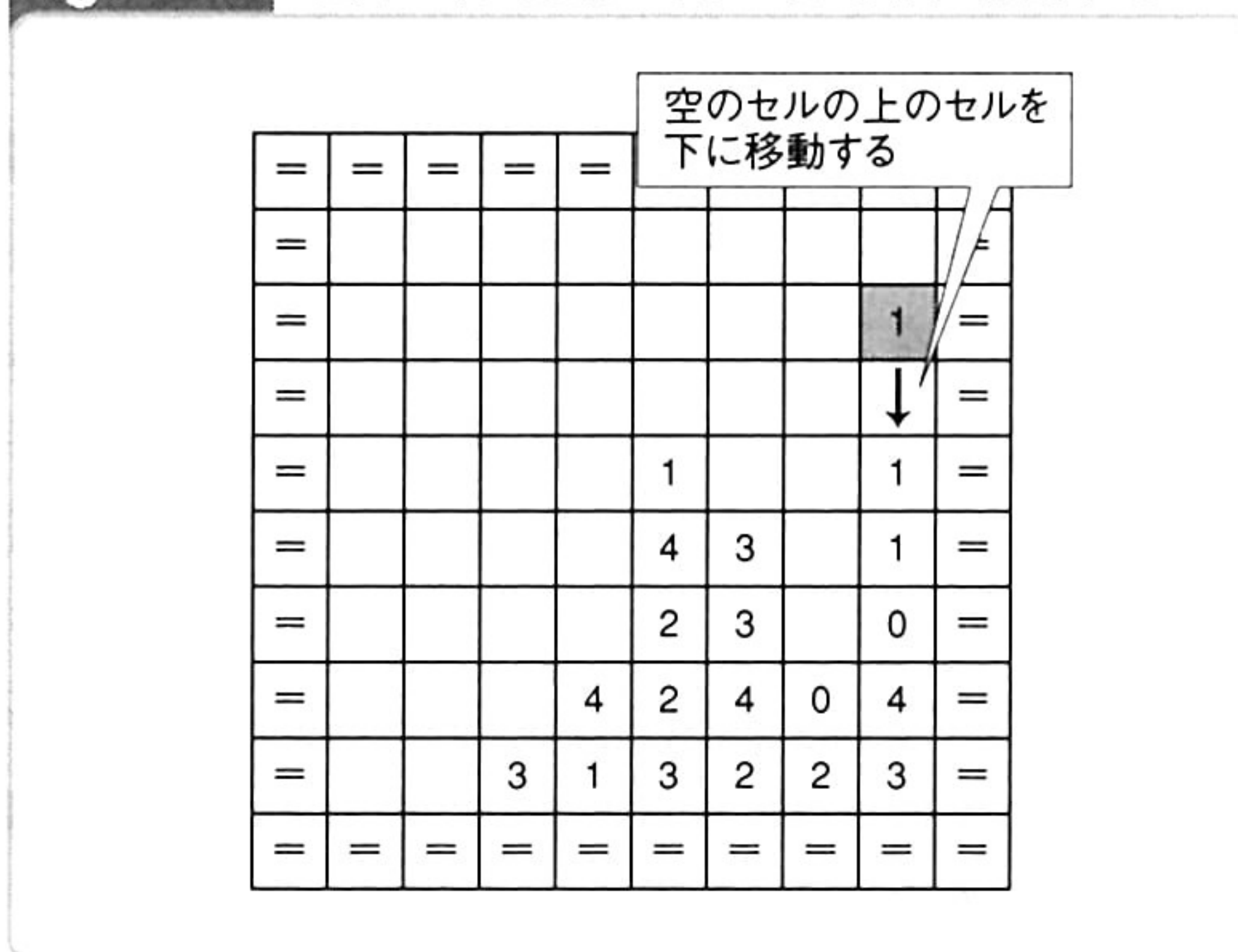
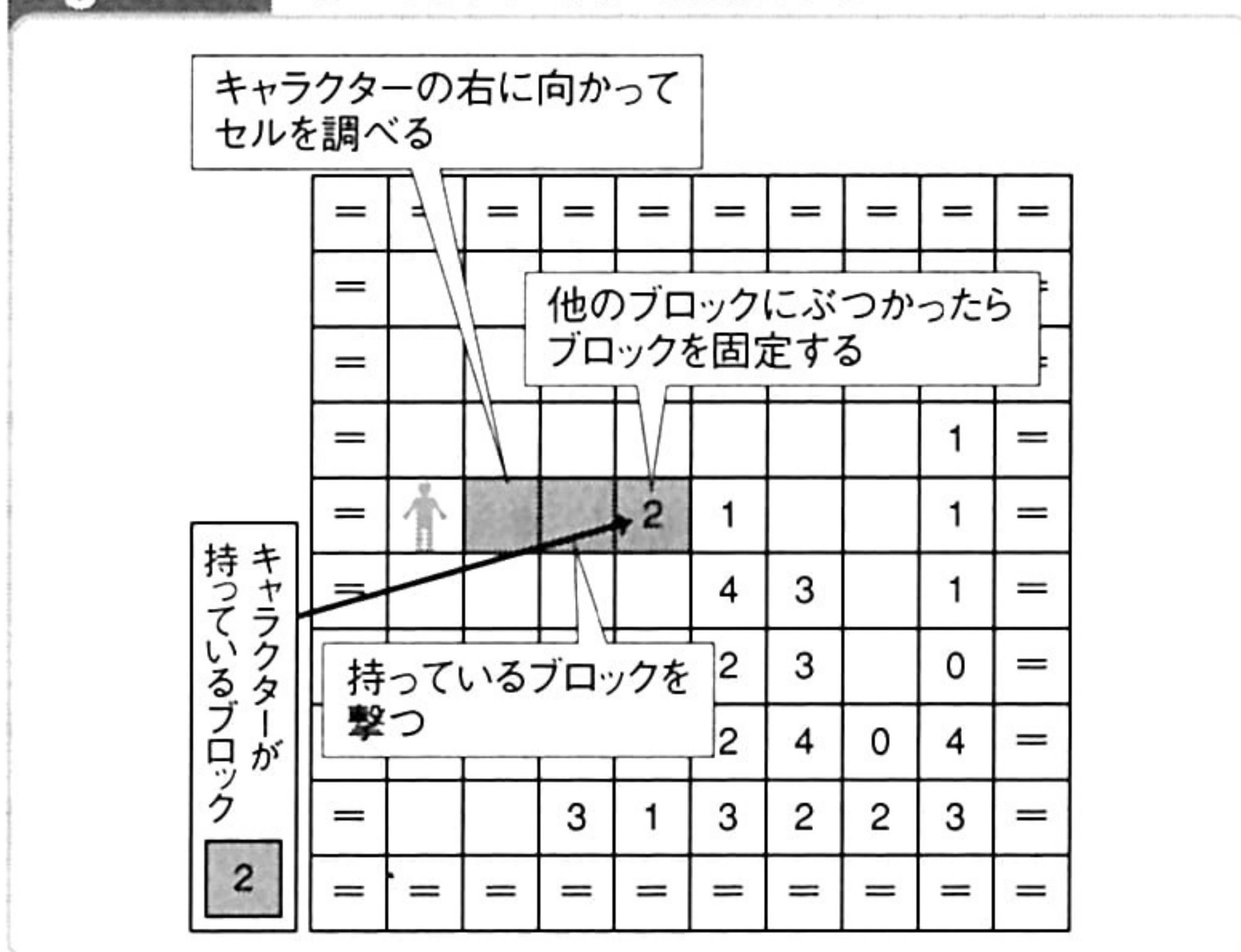
ブロックが見つかったら
セルを空にする

キャラクターが持っているブロック

Fig. 4-71 撃ったブロックを固定する

Fig. 4-72 空のセルの上にあるセルを下に移動する

Fig. 4-73 空の列の左のセルを右に移動する

Fig. 4-74 同じ種類のブロックが隣接する数を調べる

ブロックを引き寄せたときと、ブロックを撃ったときには、ステージ内の空のセルを探します。空のセルがあったら、上にあるセルを下に移動します(Fig. 4-72)。また、上端から下端まで空の列があるときには、それよりも左のセルを右に移動します(Fig. 4-73)。

ブロックの移動が終わったら、ステージ内のすべてのセルについて、同じ種類のブロックが隣接している数を調べます(Fig. 4-74)。上下左右に同じ種類のブロックがあったら、そのブロックのさらに上下左右についても、再帰的に調べます。規定数(3個)以上が隣接していたら、ブロックを消去します。

プログラム

List 4-8はブロックを引き寄せて撃つプログラムです。ステージの移動処理と、ブロックを消す処理を掲載しました。

移動処理(Move関数)は、入力状態・取得状態・射出状態・落下状態・消去判定状態・消去

状態に分かれています。入力状態では、レバー入力に応じて、キャラクターを上下に動かします。ボタンを押したら、ブロックを持っていないときには引き寄せて、持っているときには撃ちます。引き寄せる場合は取得状態に、撃つ場合には射出状態に移行します。

取得状態では、キャラクターに近づくようにブロックを動かします。ブロックがキャラクターの右隣に達したら、落下状態に移行します。

射出状態では、キャラクターから遠ざかるようにブロックを動かします。ブロックが他のブロックにぶつかったら、セルにブロックを書き込んで、ブロックを固定します。そして、落下状態に移行します。

落下状態では、空のセルの上にあるセルを、1段ずつ落下させます。また、空いた列がある場合には、その列よりも左にあるセルを、1列ずつ右に移動します。移動が終わったら、消去判定状態に移行します。

消去判定状態では、同じ種類のブロックが規定数以上隣接しているかどうかを調べ、隣接しているときにはブロックを消去します。ブロックを数えたり消したりする処理は、ブロックを消す処理（Erase関数）で行います。

ブロックを消す処理は、隣接した同じ種類のブロックを数える処理を兼ねています。上下左右のセルを再帰的に調べることで、隣接したすべてのブロックを見つけます。なお、この処理は「つながったブロックを消す」（→p. 241）のブロックを消す処理とまったく同じです。

ブロックを消したら、消去状態に移行します。消去状態では、一定時間が経過するのを待ってから、消えるブロックのセルを空にして、ブロックを完全に消します。

List 4-8 ブロックを引き寄せて撃つ（CDrawnAndShotBlockStageクラス）

```
// ステージの移動処理
bool CDrawnAndShotBlockStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 入力状態
    if (State==0) {

        // レバー入力に応じて、キャラクターを上下に動かす
        if (!PrevLever) {
            if (is->Up && CY>2) CY--; else
            if (is->Down && CY<ys-2) CY++;
        }
        PrevLever=is->Up||is->Down;

        // ボタンを押したら、ブロックを引き寄せたり撃ったりする
        if (!PrevButton && is->Button[0]) {

            // ブロックを持っていないときには引き寄せる
            if (Block==' ') {
```

```
                //キャラクターの右に向かって、
                //ブロックのセルを探す
                int x;
                for (x=3; x<xs-1 && Cell->Get(x, CY)==' '; x++) ;

                //ブロックのセルが見つかったら、
                //ブロックを引き寄せる
                if (x<xs-1) {

                    //引き寄せるブロックの種類を取得する
                    Block=Cell->Get(x, CY);

                    //セルを空にする
                    Cell->Set(x, CY, ' ');

                    //引き寄せるブロックの座標を設定する
                    BX=x;

                    //取得状態に移行する
                    State=1;
                }
            } else

            //ブロックを持っているときには撃つ
            {
                //キャラクターの右が空いているときには、
                //ブロックを撃つ
                if (Cell->Get(BX+1, CY)==' ') {

                    //射出状態に移行する
                    State=2;
                }
            }
        }
        PrevButton=is->Button[0];
    }

    //取得状態
    if (State==1) {

        //ブロックがキャラクターの右隣に達したら、
        //落下状態に移行する
        if (BX==CX+1) {
            State=3;
        } else

        //キャラクターの右隣に達するまでは、
        //ブロックを左に移動する
        {
            BX--;
```

```
        }
    }

    // 射出状態
    if (State==2) {

        // ブロックの右側のセルが空でなければ、
        // ブロックを現在の位置に固定する
        if (Cell->Get(BX+1, CY)!=' ') {

            // セルにブロックを書き込む
            Cell->Set(BX, CY, Block);

            // 手持ちのブロックを空にする
            Block=' ';

            // 落下状態に移行する
            State=3;
        } else

        // ブロックの右側のセルが空であるかぎり、
        // ブロックを右に移動する
        {
            BX++;
        }
    }

    // 落下状態
    if (State==3) {

        // ステージ内の空のセルを探す
        for (int x=3; x<xs-1; x++) {
            for (int y=2; y<ys-1; y++) {

                // 空のセルが見つかったときの処理
                if (Cell->Get(x, y)==' ') {

                    // 上にあるセルを1段ずつ落とす
                    for (int i=y; i>2; i--) {
                        Cell->Set(x, i, Cell->Get(x, i-1));
                    }

                    // 最上段のセルを空にする
                    Cell->Set(x, 2, ' ');
                }
            }
        }

        // 空の列を探す
        for (int x=3; x<xs-1; x++) {
```

```
            // 上端から下端までセルが空かどうかを調べる
            int y;
            for (y=2; y<ys-1 && Cell->Get(x, y)==' '; y++) ;

            // セルが空のときの処理
            if (y==ys-1) {

                // 空の列よりも左のセルを右に移動する
                for (int i=x; i>3; i--) {
                    for (int j=2; j<ys-1; j++) {
                        Cell->Set(i, j, Cell->Get(i-1, j));
                    }
                }

                // 左端の列は空にする
                for (int j=2; j<ys-1; j++) {
                    Cell->Set(3, j, ' ');
                }
            }
        }

        // 消去判定状態に移行する
        State=4;
    }

    // 消去判定状態
    if (State==4) {

        // ブロックが消えなかったときには、
        // 初期状態に移行する
        State=0;

        // 隣接する同じ種類のブロックを探す
        for (int y=2; y<ys-1; y++) {
            for (int x=3; x<xs-1; x++) {
                char c=Cell->Get(x, y);

                // 同じ種類のブロックが規定数以上隣接していたら、
                // ブロックを消す
                if (
                    !(c&0x80) &&
                    c!=' ' &&
                    Erase(x, y, c, 0x80)>=
                        DRAWN_AND_SHOT_BLOCK_ERASE
                ) {
                    // ブロックを消えるブロックとしてマークする
                    Erase(x, y, c|0x80, 0x40);

                    // 消去状態に移行する
```

```
                    Time=0;
                    State=5;
                }
            }
        }

        // カウントずみのマークを解除する
        for (int y=2; y<ys-1; y++) {
            for (int x=3; x<xs-1; x++) {
                Cell->Set(x, y, Cell->Get(x, y)&0x7f);
            }
        }
    }

    // 消去状態
    if (State==5) {

        // 一定時間が経過したら、消えるブロックを完全に消す
        Time++;
        if (Time==30) {

            // 消えるブロックを探す
            for (int x=3; x<xs-1; x++) {
                for (int y=2; y<ys-1; y++) {

                    // 消えるブロックを見つけたら、
                    // セルを空にする
                    if (Cell->Get(x, y)&0x40) {
                        Cell->Set(x, y, ' ');
                    }
                }
            }

            // 落下状態に移行する
            State=3;
        }
    }
    return true;
}

// ブロックを消す処理
// 隣接した同じ種類のブロックを数える処理も兼ねている
int CDrawnAndShotBlockStage::Erase(int x, int y, char c, int mask) {

    // 現在位置のセルが指定された種類のセルならば、
    // セルをマークする
    if (Cell->Get(x, y)==c) {

        // 指定されたビットをセットすることによって、セルをマークする
```

```
        Cell->Set(x, y, c|mask);

        // 上下左右のセルについても再帰的に調べて、
        // 隣接した同じ種類のセルの数を返す
        return
            1+
            Erase(x-1, y, c, mask)+
            Erase(x+1, y, c, mask)+
            Erase(x, y-1, c, mask)+
            Erase(x, y+1, c, mask);
    }

    // 現在位置のセルは指定された種類のセルではないので、
    // 隣接した同じ種類のセルの数としては0を返す
    return 0;
}
```

SAMPLE

「DRAWN AND SHOT BLOCK」は「ブロックを引き寄せて撃つ」のサンプルです。

レバーの上下(カーソルキーの上下)でキャラクターを動かします。ボタン0(Zキー)を押すと、キャラクターと同じ高さにあるブロックを引き寄せることができます。もう一度ボタンを押すと、引き寄せたブロックを撃ちます。撃ったブロックは右に飛んでいき、他のブロックにぶつかると落ちます。

ブロックを引き寄せたときや撃ったときに、同じ種類のブロックが規定数(3個)以上隣接していると、ブロックが消えます。消えたブロックの上にあったブロックは落下します。また、すべてのブロックが消えた列ができると、その列よりも左にある列が右に移動して、空いた列を詰めます。

DRAWN AND SHOT BLOCK → p. 388

ブロックを突き落として集める

ステージに並んだブロックを突き落として、別の領域に集めるアクションです。同じ種類のブロックだけを集めると、消すことができます。

ステージには多数のブロックが積まれています(Fig. 4-75)。ステージの右端にはキャラクターがいます。キャラクターはレバー入力で上下に動かすことができます。

ボタンを押すと、キャラクターと同じ高さにあるブロックのうち、ステージ左端にあるものを突き落とすことができます(Fig. 4-76)。突き落とされたブロックは、ステージ下方にある別の領域に集められます。キャラクターと同じ高さの段は、1列左に動きます。上にあるブロックの支えがなくなった場合には、ブロックが落下してきます。

同じ種類のブロックを規定数(ここでは3個)集めると、ブロックを消すことができます

Fig. 4-75 ステージの構成

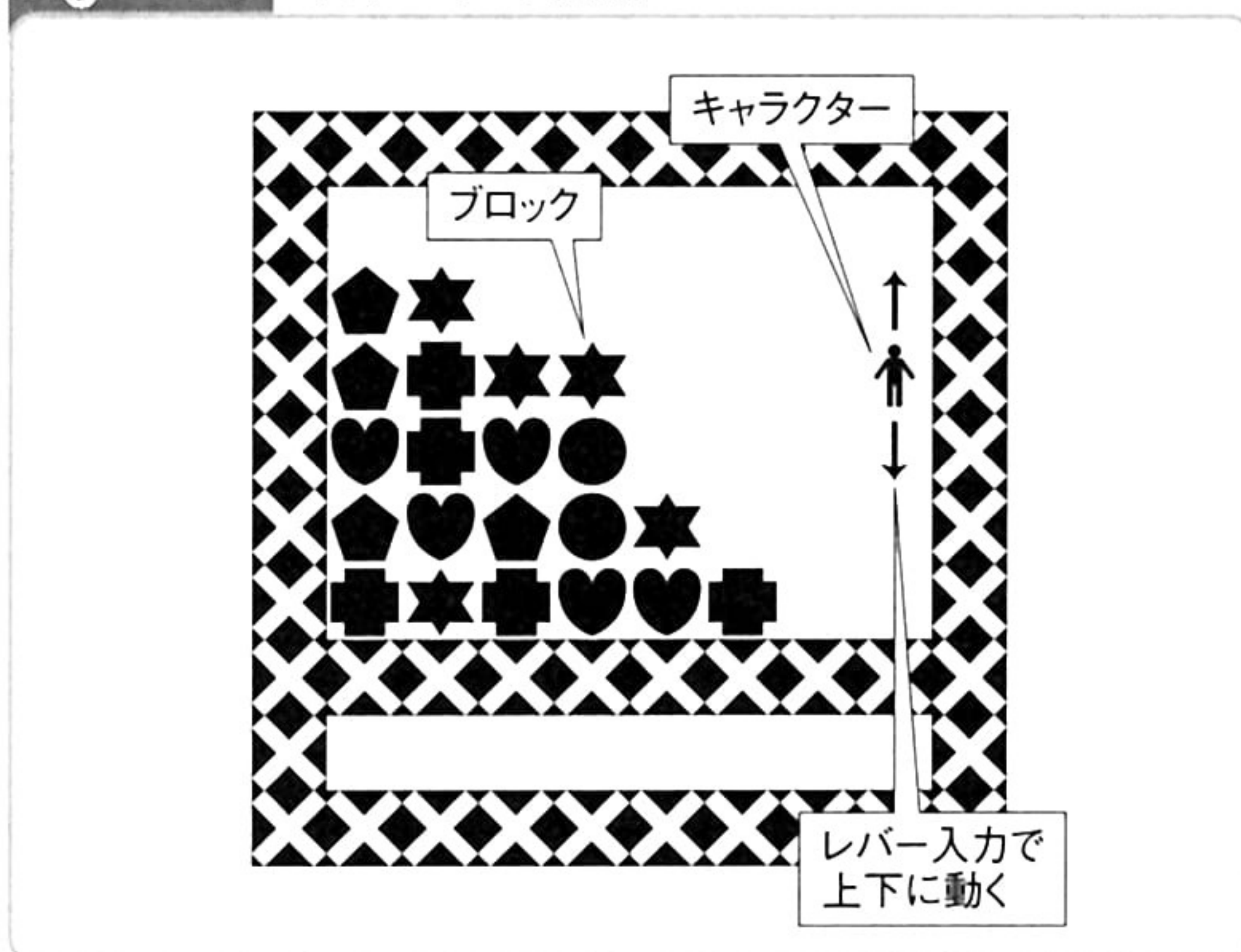
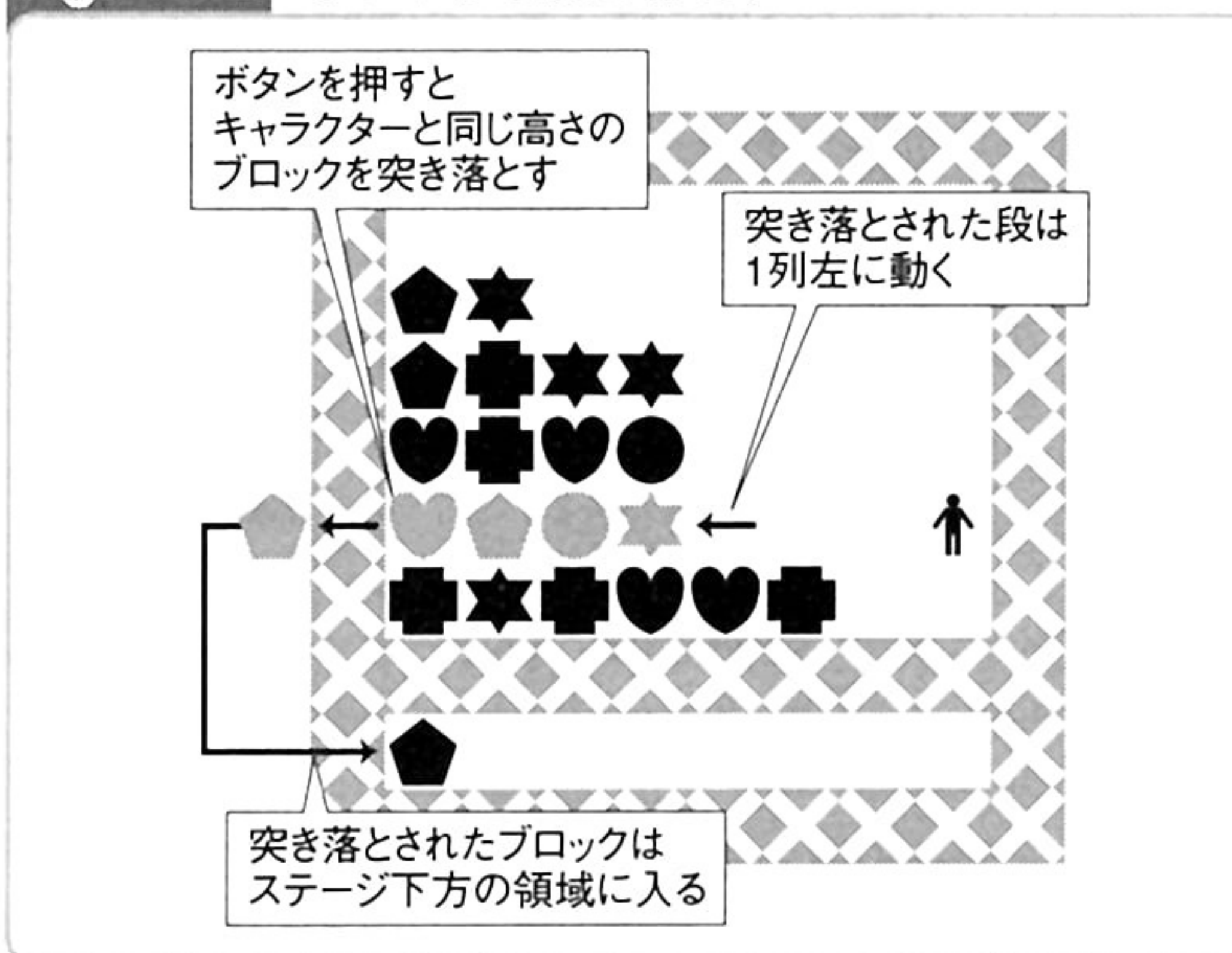
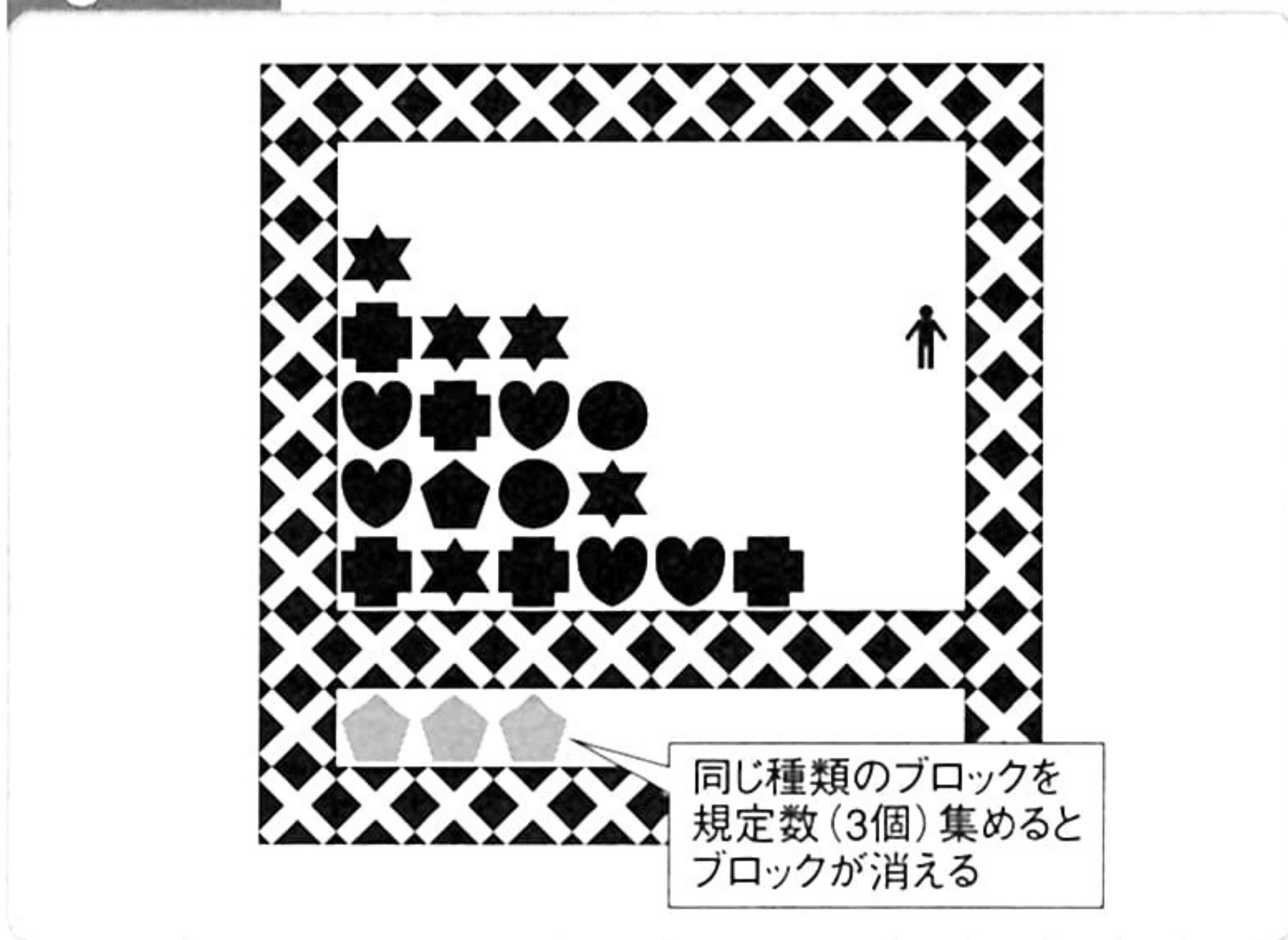
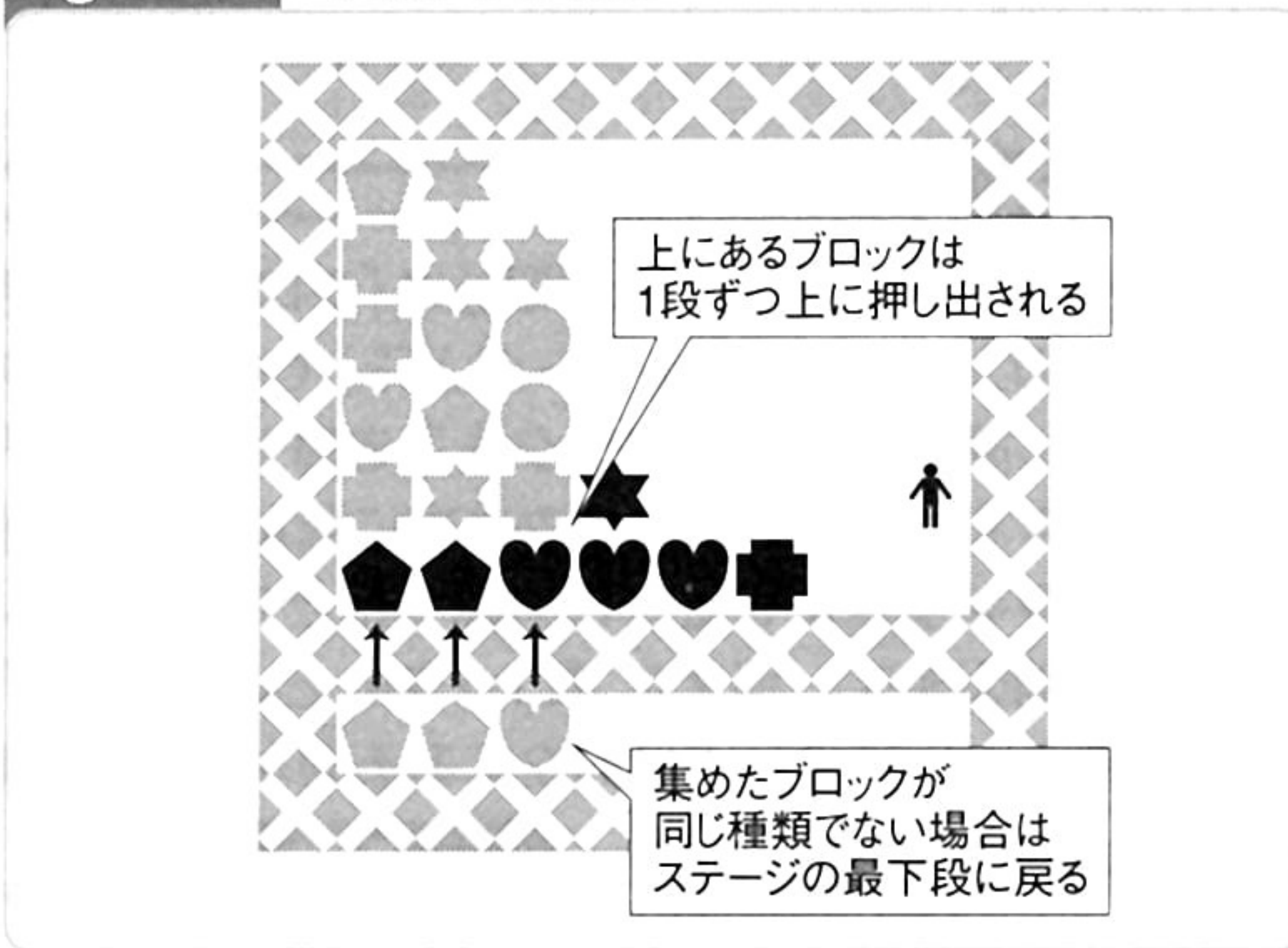
キャラクター
ブロック
レバー入力で
上下に動く

Fig. 4-76 ブロックを突き落とす

ボタンを押すと
キャラクターと同じ高さの
ブロックを突き落とす
突き落とされた段は
1列左に動く
突き落とされたブロックは
ステージ下方の領域に入る

Fig. 4-77 ブロックを消す

同じ種類のブロックを
規定数（3個）集めると
ブロックが消える

Fig. 4-78 ブロックが戻る

上にあるブロックは
1段ずつ上に押し出される
集めたブロックが
同じ種類でない場合は
ステージの最下段に戻る

(Fig. 4-77)。集めたブロックが同じ種類ではない場合には、ブロックはステージ上方の領域に戻ります (Fig. 4-78)。ブロックは積まれたブロックの最下段に入り、上にあるブロックは1段ずつ上に押し出されます。

ブロックを突き落として集めるゲームには『パズルスター』があります。このゲームでは、ステージにあるブロックを突き落として集め、時間内にすべてのブロックを消すことが目的です。ブロックのなかには、特定の順番で集めなければならないものもあるため、ブロックを突き落とす順番をよく考える必要があります。

アルゴリズム

ブロックを突き落として集めるアクションを実現するには、まずステージをセルで表現します (Fig. 4-79)。ステージの壁は「=」で、ブロックは「0〜4」の数字で表しました。ステージ上方はブロックを積む領域で、ステージ下方は突き落としたブロックを集める領域です。

Fig. 4-79 ステージをセルで表現する

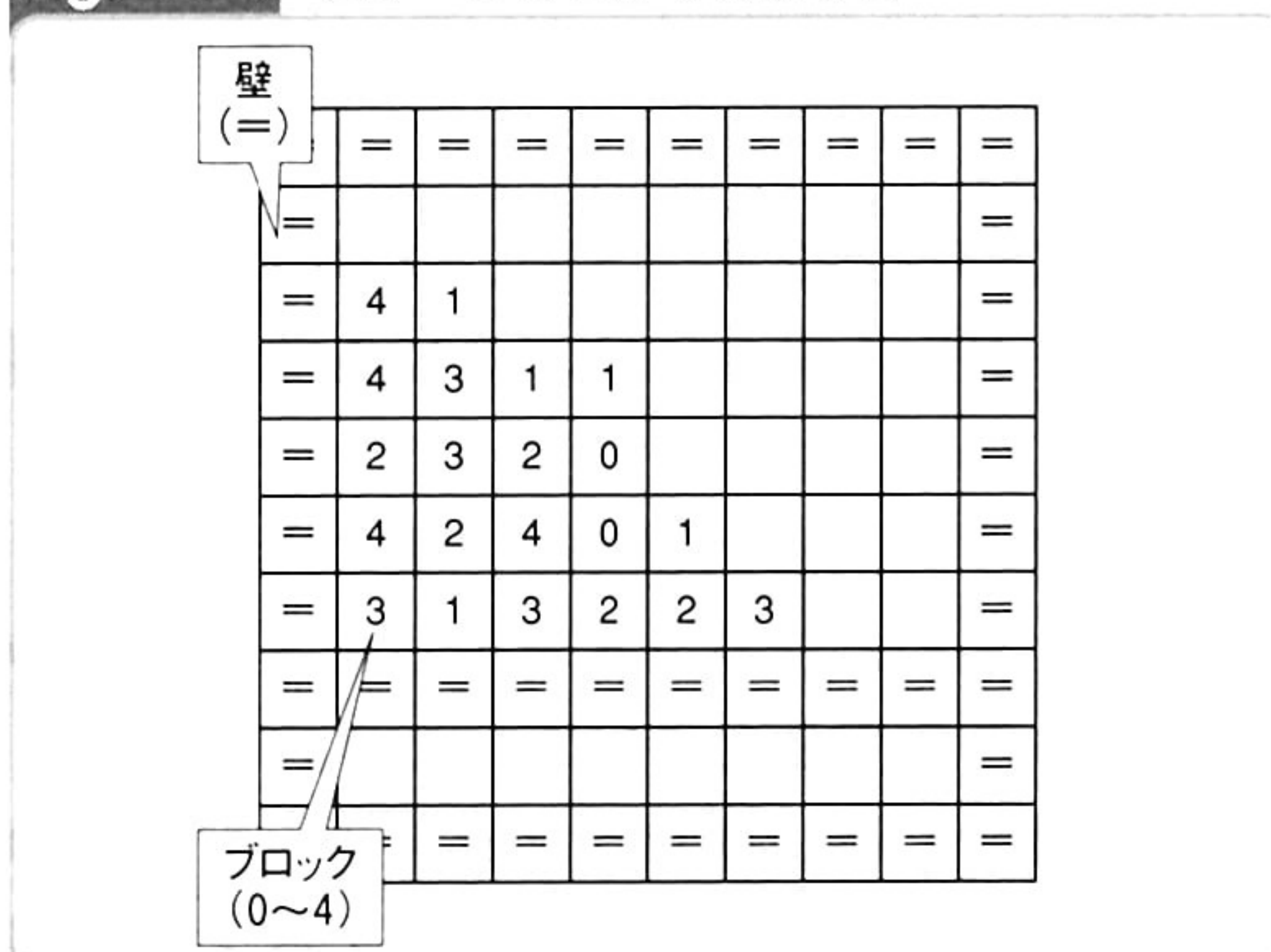



Fig. 4-80 左端のブロックを突き落とす

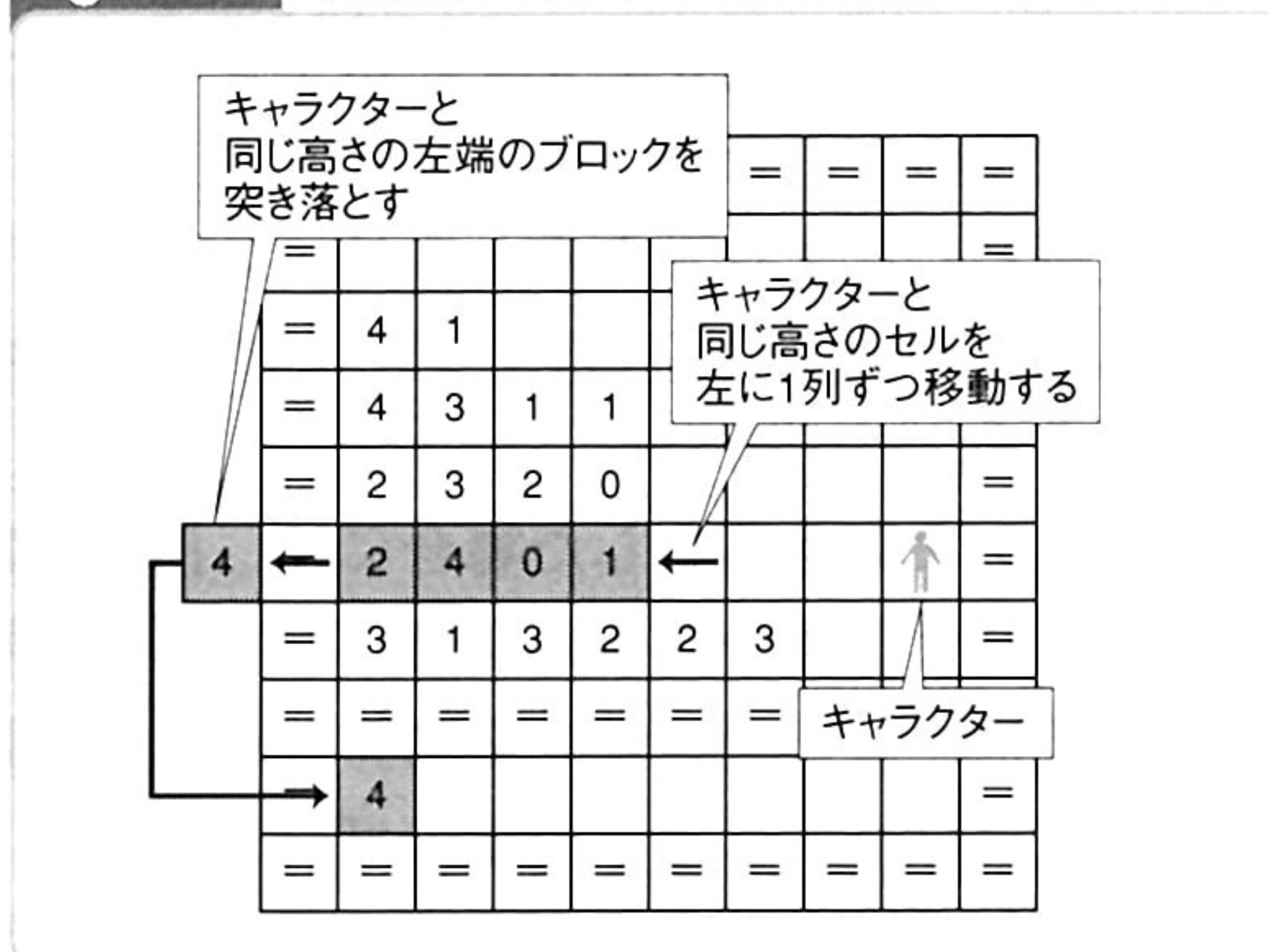



Fig. 4-81 ブロックをステージ上方に戻す

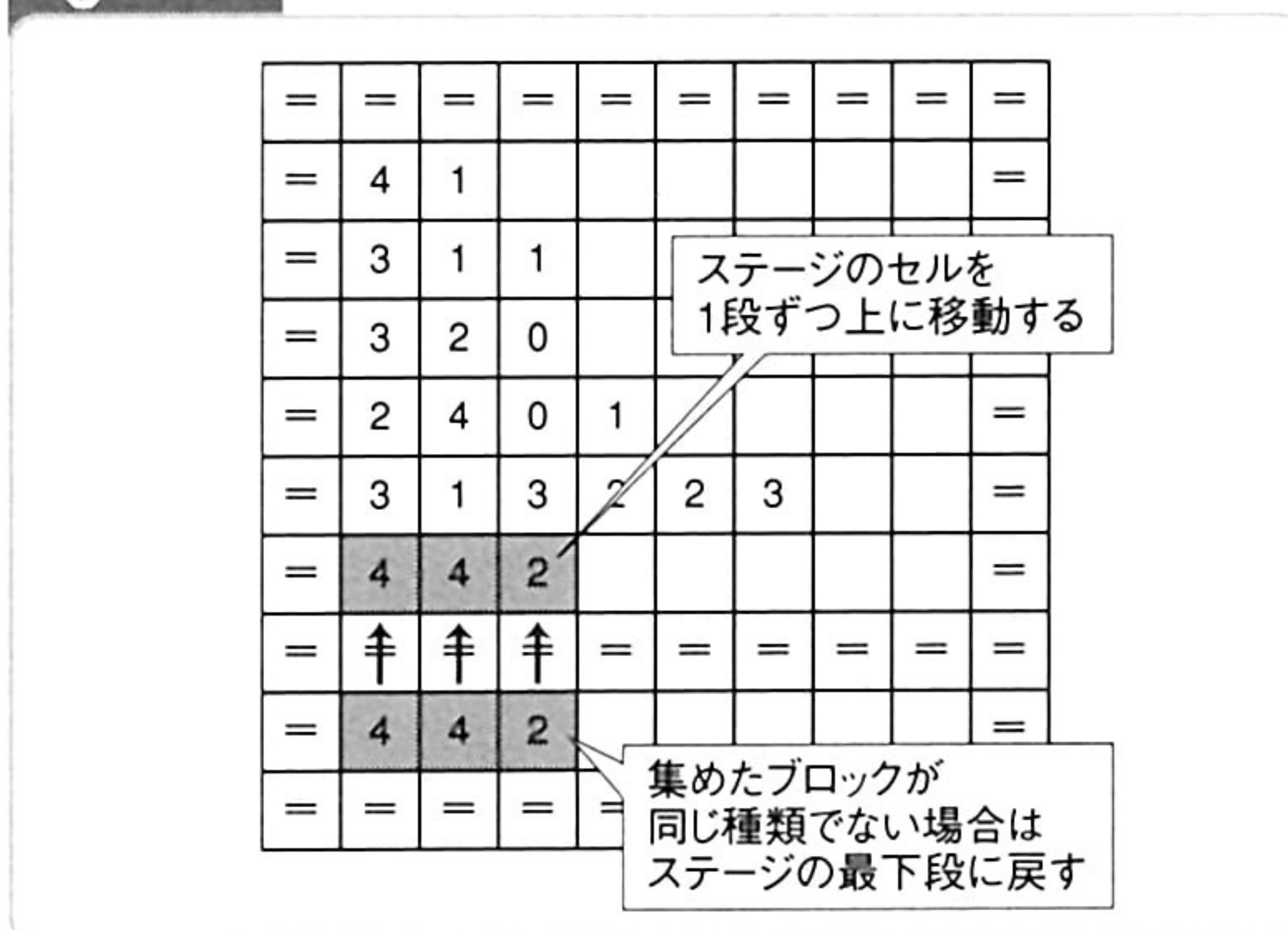
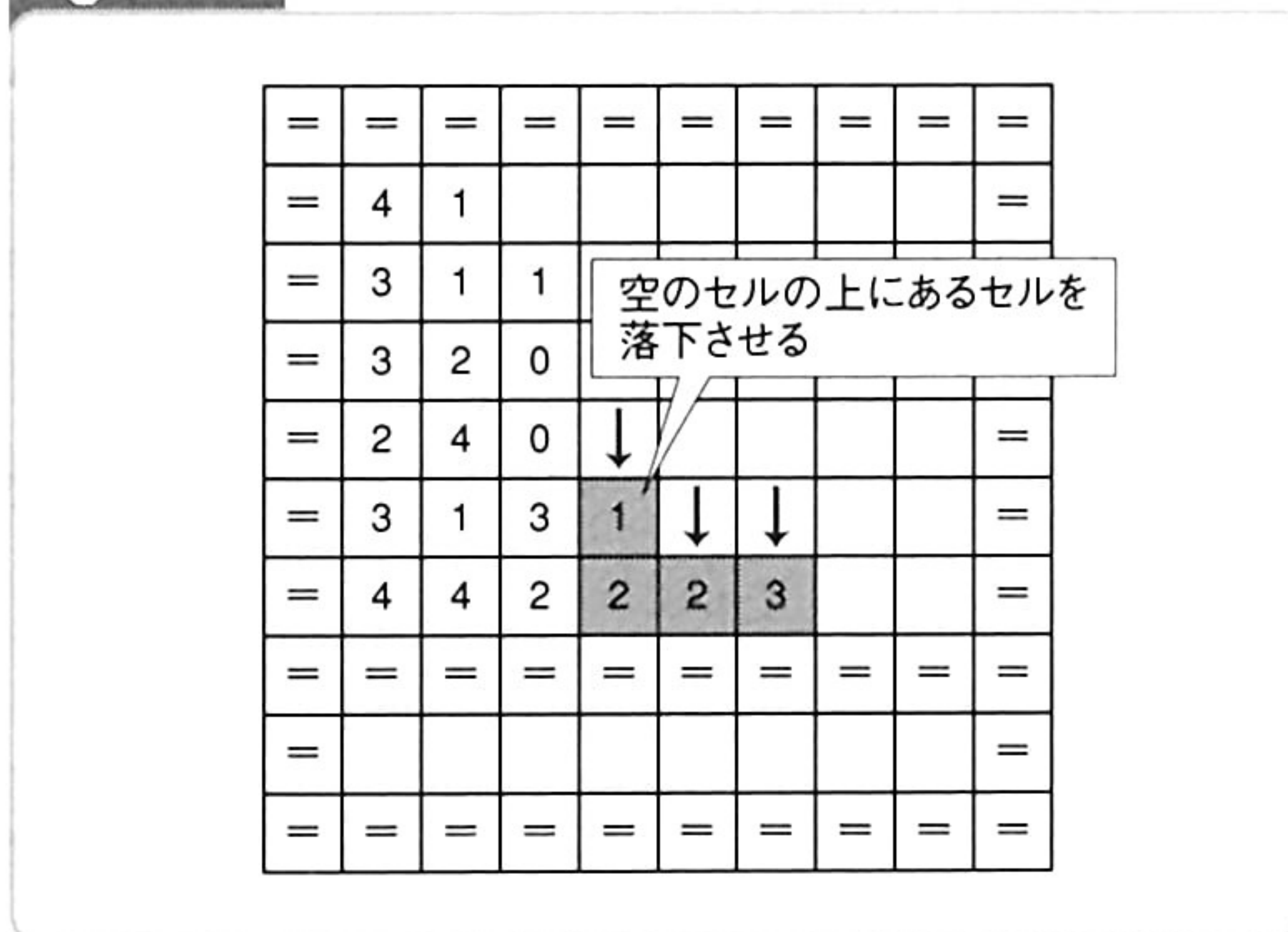



Fig. 4-82 空のセルの上にあるセルを落とす

ボタンを押したら、キャラクターと同じ高さにあるブロックのなかで、左端のものを突き落とします (Fig. 4-80)。左端のセルを取り除いて、ブロックを集める領域に移動します。また、キャラクターと同じ高さにあるセルを、左に1列ずつ移動します。

ブロックを規定数 (3個) 集めたら、同じ種類のブロックかどうかを調べます。同じ種類のブロックならば、ブロックを消します。違う種類のブロックが混じっていたら、ブロックをステージ上方に戻します (Fig. 4-81)。ステージのセルを1段ずつ上に移動し、集めたブロックを最下段に戻します。

最後に、空のセルの上にあるセルを落下させれば完了です (Fig. 4-82)。なお、ブロックを突き落としたときにも、空のセルが生じるので、上にあるセルを落とす処理を行います。

プログラム

List 4-9はブロックを突き落として集めるプログラムです。ステージの移動処理を掲載しました。

移動処理は入力状態と消去状態に分かれています。入力状態では、レバー入力に応じてキャラクターを上下に動かします。ボタンを押したら、キャラクターと同じ高さにあるブロックのうち、左端のものを突き落とします。

突き落としたブロックのセルは、ステージ下方の領域に移動します。また、ブロックを突き落とした段のセルは、1列ずつ左に移動します。空のセルがある場合には、上にあるセルを落下させます。

ブロックを突き落とした場合には、消去状態に移行します。消去状態では、集めたブロックがすべて同じ種類かどうかを調べます。同じ種類ならば、集めたブロックを消します。実際のゲームでは、ここでスコアを加算したり、敵に攻撃を送ったりするとよいでしょう。

集めたブロックに違う種類のブロックが混じっていた場合には、ブロックをステージ上方の領域に戻します。ステージのセルを1段ずつ上に移動してから、集めたブロックを最下段に追加します。

List 4-9　ブロックを突き落として集める(CPushedAndCollectedBlockStageクラス)

```
// ステージの移動処理
bool CPushedAndCollectedBlockStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 入力状態
    if (State==0) {

        // レバー入力に応じて、キャラクターを上下に動かす
        if (!PrevLever) {
            if (is->Up && CY>2) CY--; else
            if (is->Down && CY<ys-4) CY++;
        }
        PrevLever=is->Up||is->Down;

        // ボタンを押したら、
        // キャラクターと同じ高さにあるブロックを突き落とす
        if (!PrevButton && is->Button[0]) {

            // キャラクターと同じ高さにあるセルのうち、
            // 左端のセルを取得する
            char c=Cell->Get(1, CY);

            // セルが空でなければ、ブロックを突き落とす
            if (c!=' ') {

                // 集めたブロックを数える
                int x;
                for (x=1; Cell->Get(x, ys-2)!=' '; x++) ;

                // 集めたブロックの右端に、
```

```
                // 新しいブロックを配置する
                Cell->Set(x, ys-2, c);

                // 規定数(3個)のブロックを集めたら、
                // 消去状態に移行する
                if (x==PUSHED_AND_COLLECTED_BLOCK_ERASE) {
                    Time=0;
                    State=1;
                }

                // ブロックを突き落とした段のセルを、
                // 左に1列ずつ移動させる
                for (x=1; x<xs-2; x++) {
                    Cell->Set(x, CY, Cell->Get(x+1, CY));
                }
            }
        }
        PrevButton=is->Button[0];

        // 空のセルを探す
        for (int x=1; x<xs-1; x++) {
            for (int y=2; y<ys-3; y++) {

                // 空のセルを見つけたら、上にあるセルを落とす
                if (Cell->Get(x, y)==' ') {

                    // 空のセルの上にあるセルを、
                    // 1段ずつ下に移動する
                    for (int i=y; i>2; i--) {
                        Cell->Set(x, i, Cell->Get(x, i-1));
                    }

                    // 最上段のセルは空にする
                    Cell->Set(x, 2, ' ');
                }
            }
        }
    }

    // 消去状態
    if (State==1) {

        // 一定時間が経過したら、集めたブロックを調べる
        Time++;
        if (Time==30) {

            // 集めたブロックがすべて同じ種類かどうかを調べる
            char c=Cell->Get(1, ys-2);
            int x, y;
            for (x=2; Cell->Get(x, ys-2)==c; x++) ;
```

```
            // すべて同じ種類でなければ、
            // 集めたブロックをステージ上方の領域に戻す
            if (x<1+PUSHED_AND_COLLECTED_BLOCK_ERASE) {

                // ステージのセルを1段ずつ上に移動する
                for (y=2; y<ys-4; y++) {
                    for (x=1; x<xs-1; x++) {
                        Cell->Set(x, y, Cell->Get(x, y+1));
                    }
                }

                // 最下段に集めたブロックを移動する
                for (x=1; x<xs-1; x++) {
                    Cell->Set(x, ys-4, Cell->Get(x, ys-2));
                }
            }

            // ブロックを集める領域を空にする
            for (x=1; x<xs-1; x++) {
                Cell->Set(x, ys-2, ' ');
            }

            // 入力状態に移行する
            State=0;
        }
    }
    return true;
}
```

SAMPLE

「PUSHED AND COLLECTED BLOCK」は「ブロックを突き落として集める」のサンプルです。

レバーの上下（カーソルキーの上下）でキャラクターが動きます。ボタン0（Zキー）を押すと、キャラクターと同じ高さにあるブロックを突き落とすことができます。

突き落としたブロックは、ステージ下方の領域に集まります。同じ種類のブロックを規定数（3個）集めると、ブロックは消えます。同じ種類ではないブロックを集めた場合には、ブロックはステージ上方の領域に戻ります。

PUSHED AND COLLECTED BLOCK → p. 388

落ちてくるブロックを拾って積む

キャラクターを動かして、落ちてくるブロックを拾うアクションです。拾ったブロックを積み直し、同じ種類のブロックを並べることによって、ブロックを消すことができます。

ステージ上方からブロックが落ちてきます（Fig. 4-83）。ブロックにはいくつかの種類があります。本書のサンプルでは、ブロックの種類を形で区別していますが、実際のゲームでは色で区別してもよいでしょう。

ステージ中央にはキャラクターがいます。キャラクターはレバー入力で左右に動かすことができます。ステージ下方の灰色で示した領域は、拾ったブロックを積むための領域です。

落ちてくるブロックの下にキャラクターを移動すると、ブロックを拾うことができます（Fig. 4-84）。拾ったブロックは灰色で示しました。キャラクターはブロックを持ったまま、左右に移動することができます。本書のサンプルでは、一度に持ち運べるブロックは1個だけです。

ブロックを持っているときにボタンを押すと、キャラクターの真下にブロックを積むことができます（Fig. 4-85）。ブロックを積めるのは、ステージ下方の灰色で示した領域のなかだけです。

ブロックを上手に積んで、縦・横・斜めのいずれかに同じ種類のブロックを規定数（ここでは3個）以上並べると、ブロックを消すことができます。ブロックが消えると、その上にあったブロックが落ちていきます（Fig. 4-86）。そして、再び同じ種類のブロックが並ぶと、ブロックが連鎖的に消えます。

落ちてくるブロックを拾って積むゲームには『クラックス』があります。このゲームでは、落ちてくるブロックを、画面奥から手前に向かって転がってくるように表現しています。ちょうど、ベルトコンベアの終点で、流れてくるブロックを拾っているような状況です。

このゲームでは、一度に複数のブロックを拾うことができます。ボタンを押すと、拾ったブロックを画面下方に積むことができます。複数のブロックを拾ったときには、拾った順番とは逆の順番でブロックを積むので、注意が必要です。ブロックを上手に積んで、同じ種類のブロックを縦・横・斜めに並べると、ブロックを消すことができます。

Fig. 4-83 ステージの構成

ブロック
キャラクター
レバー入力で
左右に動く
時間とともに
少しずつ落下する
拾ったブロックを
積むための領域
（灰色の部分）

Fig. 4-84 ブロックを拾う

落ちてくるブロックの下に
キャラクターを移動して
ブロックを拾う
拾ったブロックを
持ったまま
左右に移動できる

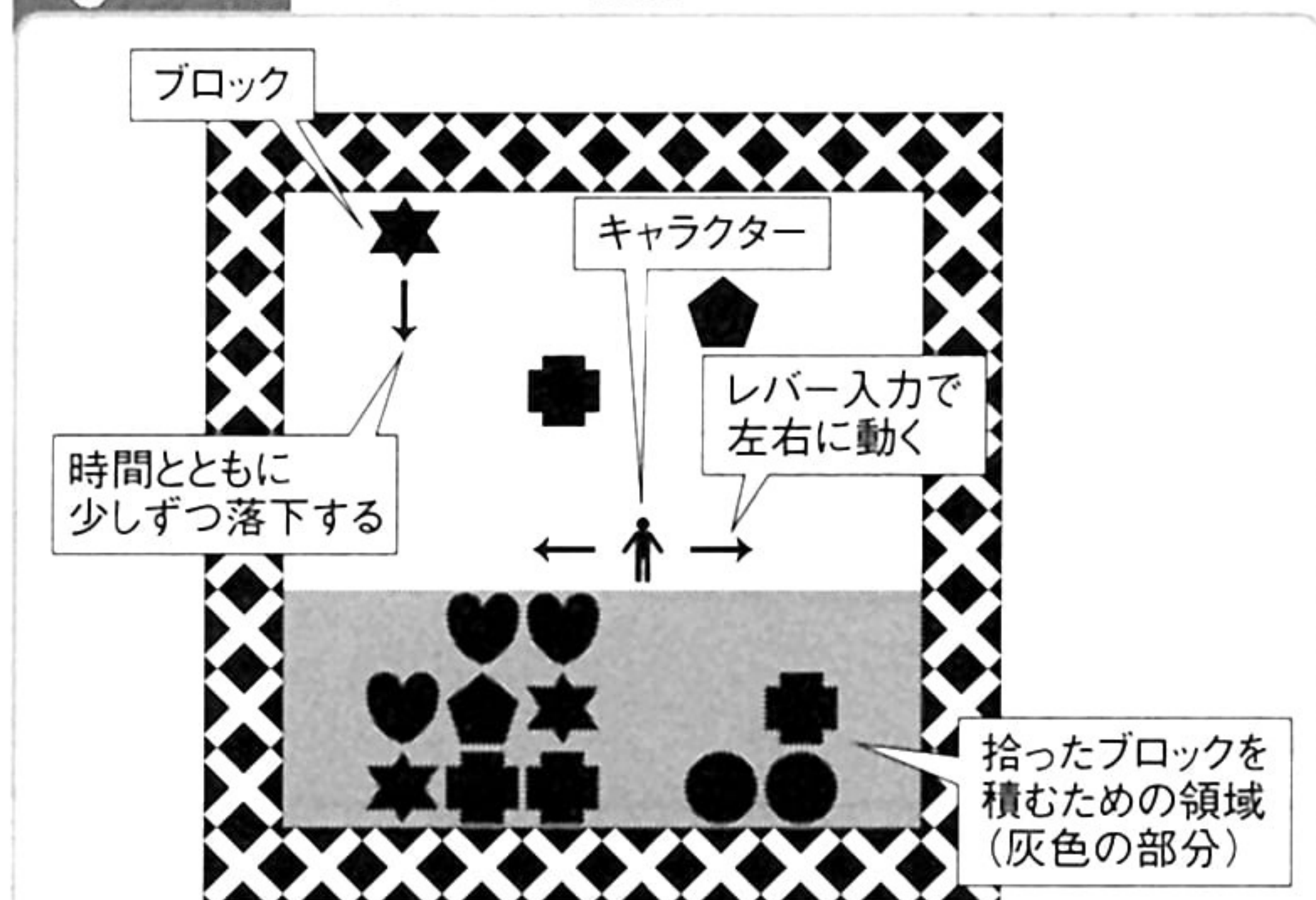
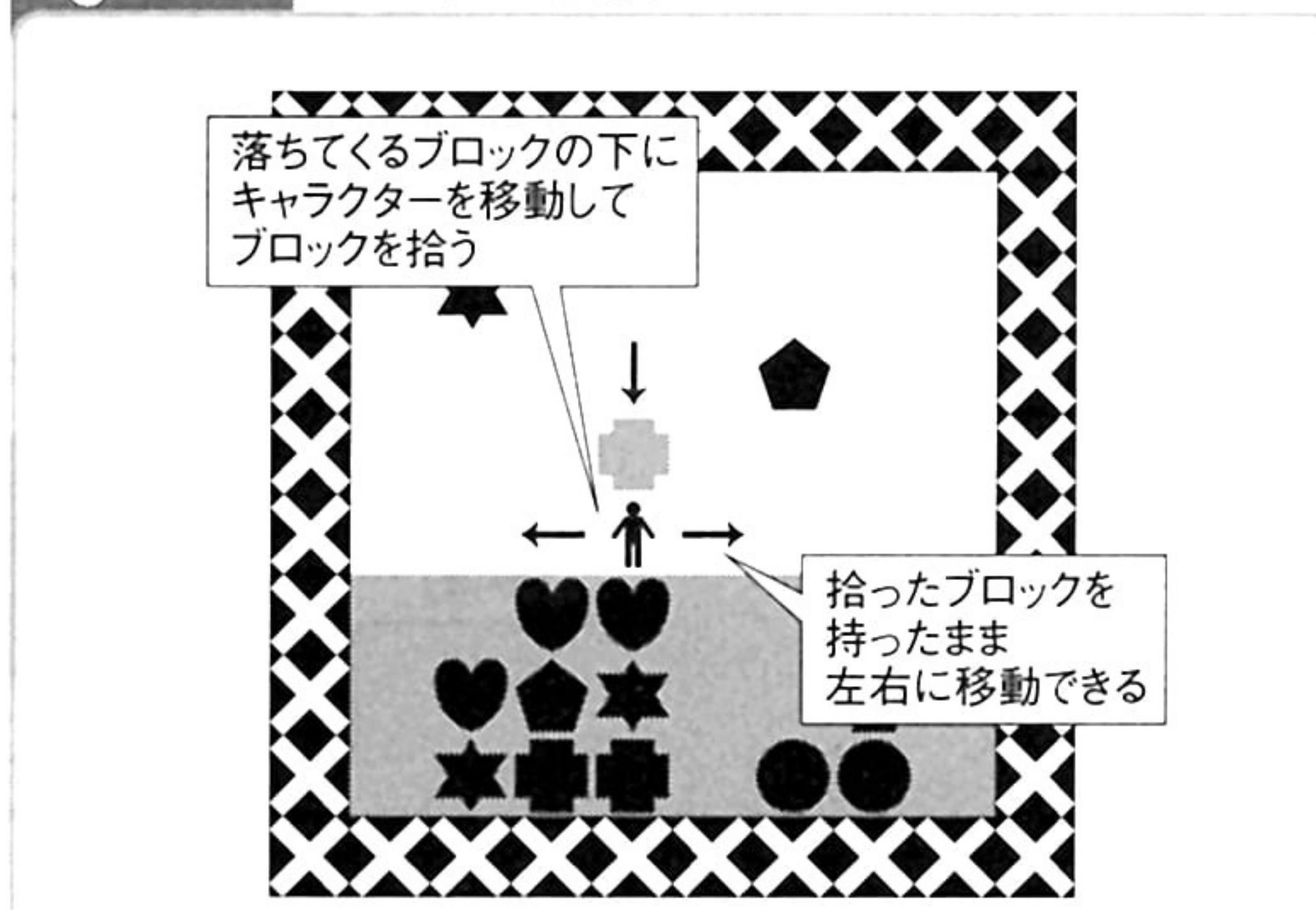



Fig. 4-85 拾ったブロックを積む

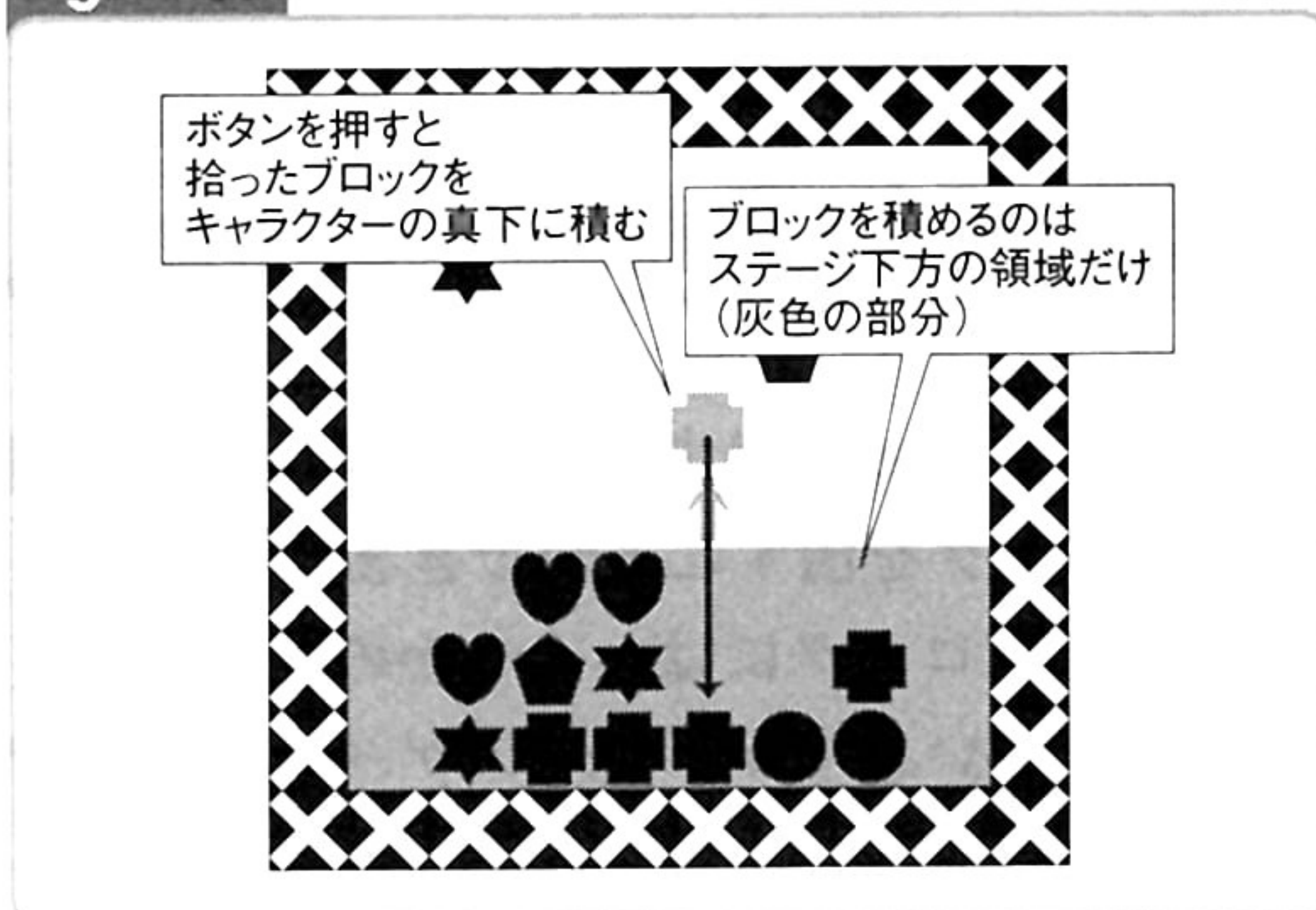
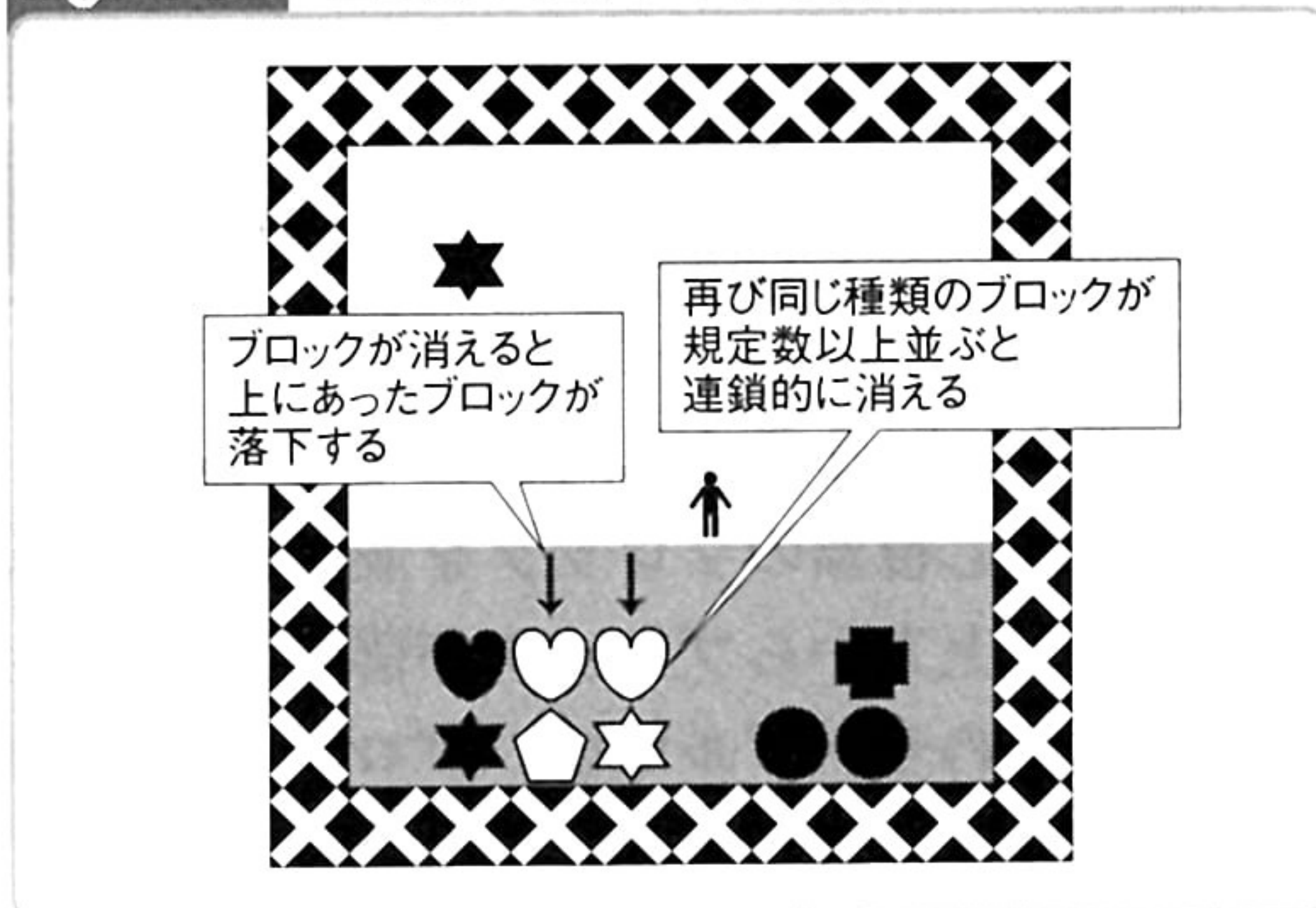



Fig. 4-86 上にあったブロックが落ちる

アルゴリズム

落ちてくるブロックを拾って積むアクションは、拾うアクションと、積むアクションの組み合わせです。まずはステージをセルで表現します（Fig. 4-87）。ステージの壁は「＝」で、ブロックは「0～4」の数字で表しました。

一定時間ごとに、ステージ上方のセルを1段ずつ下に移動することによって、ブロックを落とします（Fig. 4-88）。ブロックが最下段まで落ちたときには、本書のサンプルでは単にブロックを消しますが、実際のゲームではミスにすることが多いでしょう。

ブロックを落としたときに、ブロックの真下にキャラクターがいたら、キャラクターはブロックを拾います（Fig. 4-89）。拾ったブロックのセルを空にして、ブロックの種類を記録しておきます。

キャラクターがブロックを持っているときにボタンを押したら、ブロックをステージ下方に積みます（Fig. 4-90）。キャラクターの真下のセルに、記録しておいたブロックを書き込みます。

次に、ステージ下方のセルについて、空のセルの上にあるセルを下に移動します（Fig. 4-91）。

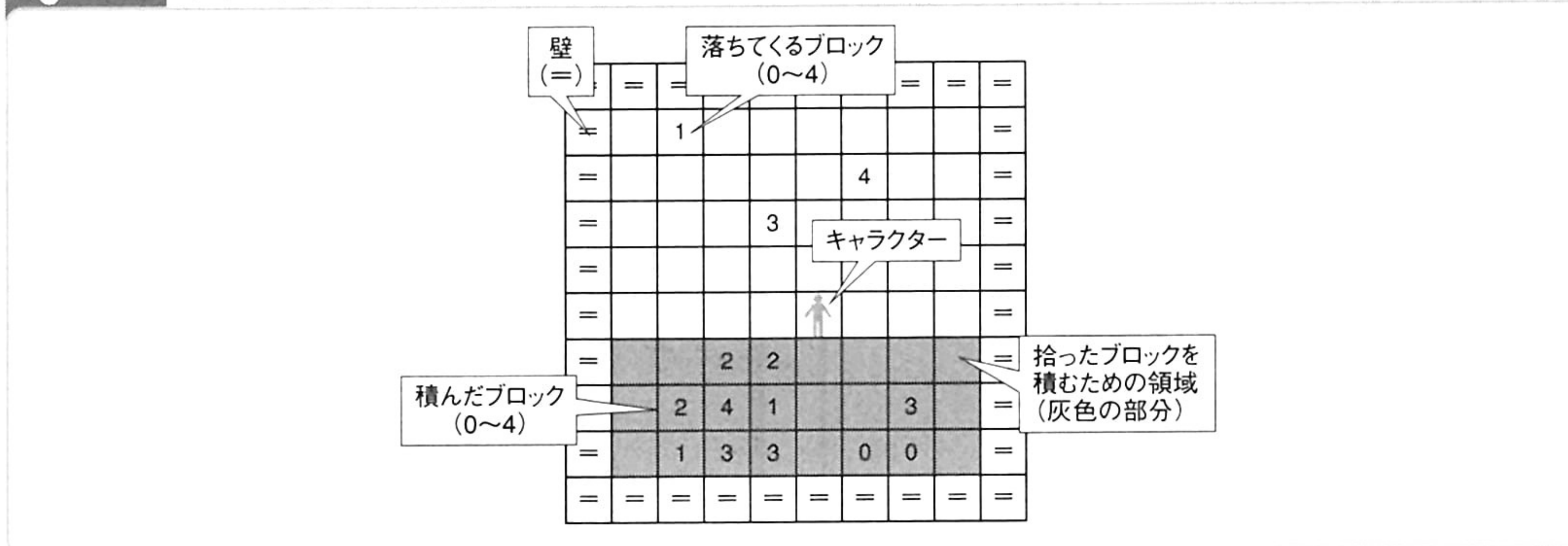
Fig. 4-87 ステージをセルで表現する

キャラクターの真下のセルに書き込んだブロックも落下して、適切な場所に着地します。
ブロックが落下したら、同じ種類のブロックが並んでいるかどうかを調べます（Fig. 4-92）。

Fig. 4-88　ブロックを落とす

ステージ上方のセルを1段ずつ下に移動してブロックを落とす
ステージ下方のセルは動かさない

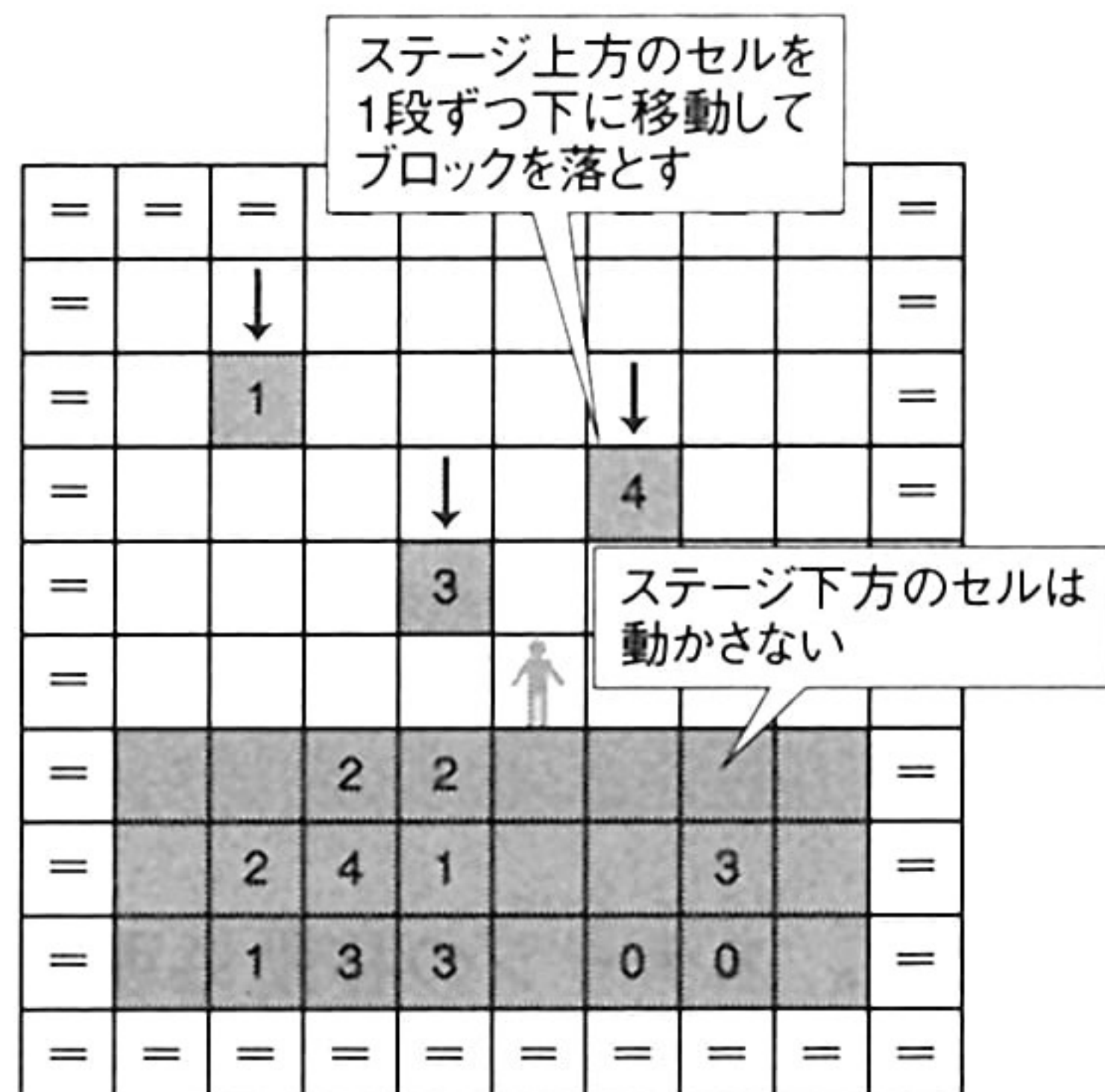
Fig. 4-89　落ちてきたブロックを拾う

拾ったブロックのセルは空にする
拾ったブロックの種類を記録する
キャラクターが持っているブロック
ブロックの真下にキャラクターがいたらブロックを拾う

Fig. 4-90　ブロックをステージ下方に積む

ボタンを押したら持っているブロックをキャラクターの真下に書き込む
キャラクターが持っているブロック

Fig. 4-91　ステージ下方のセルを落とす

ステージ下方について空のセルの上にあるセルを落下させる
キャラクターの真下に書き込んだブロックも適切な場所に落下する

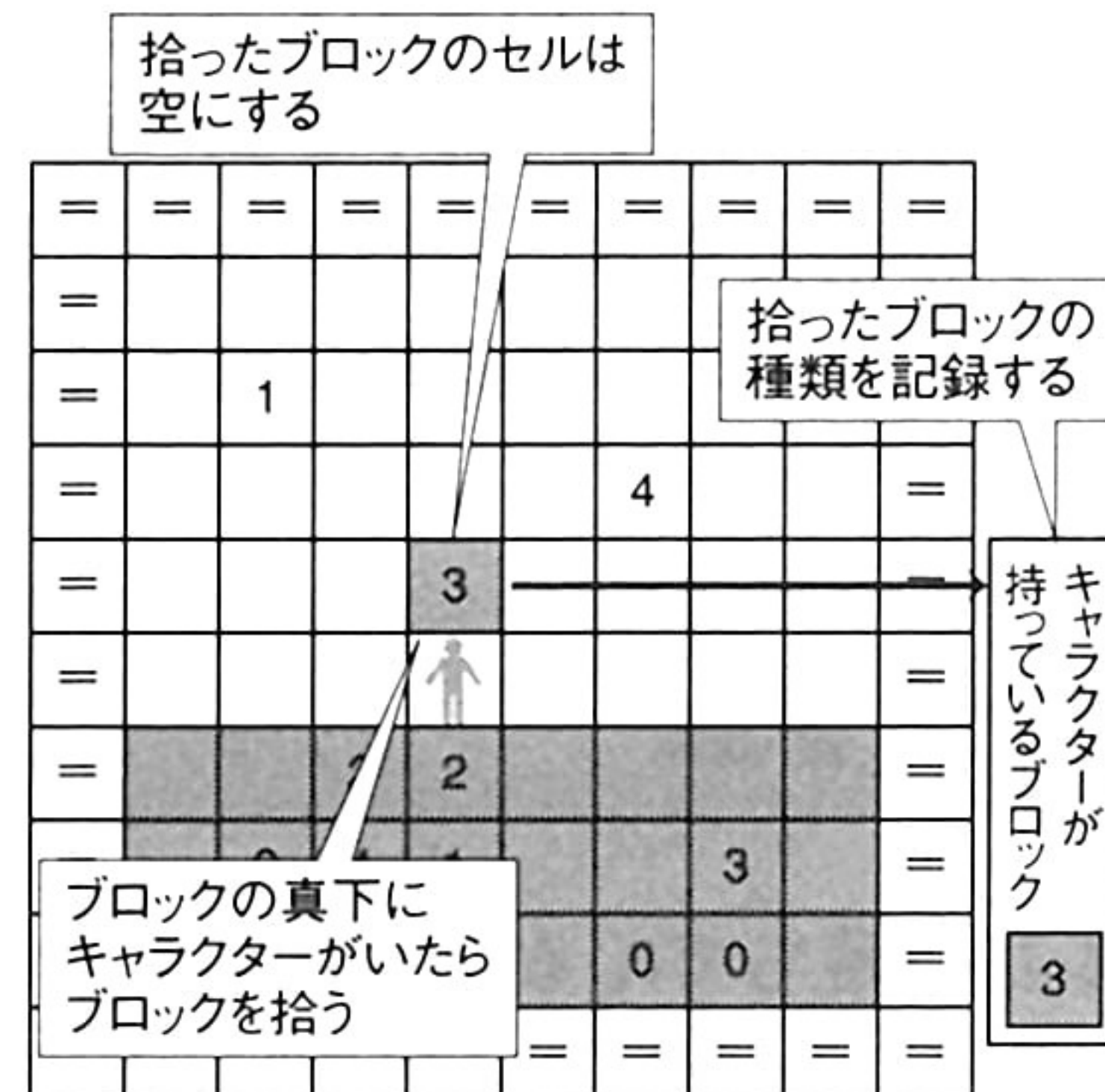
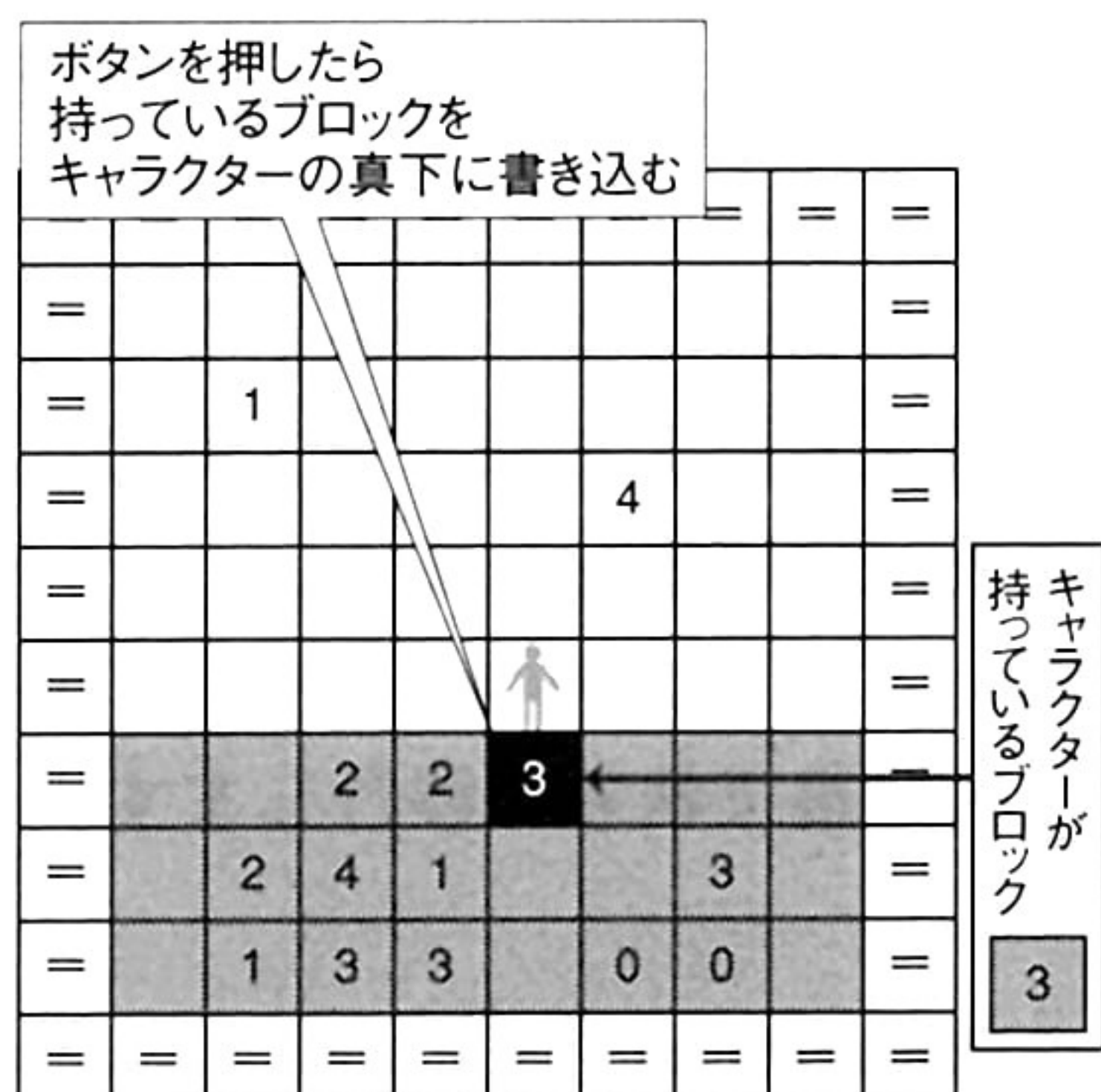
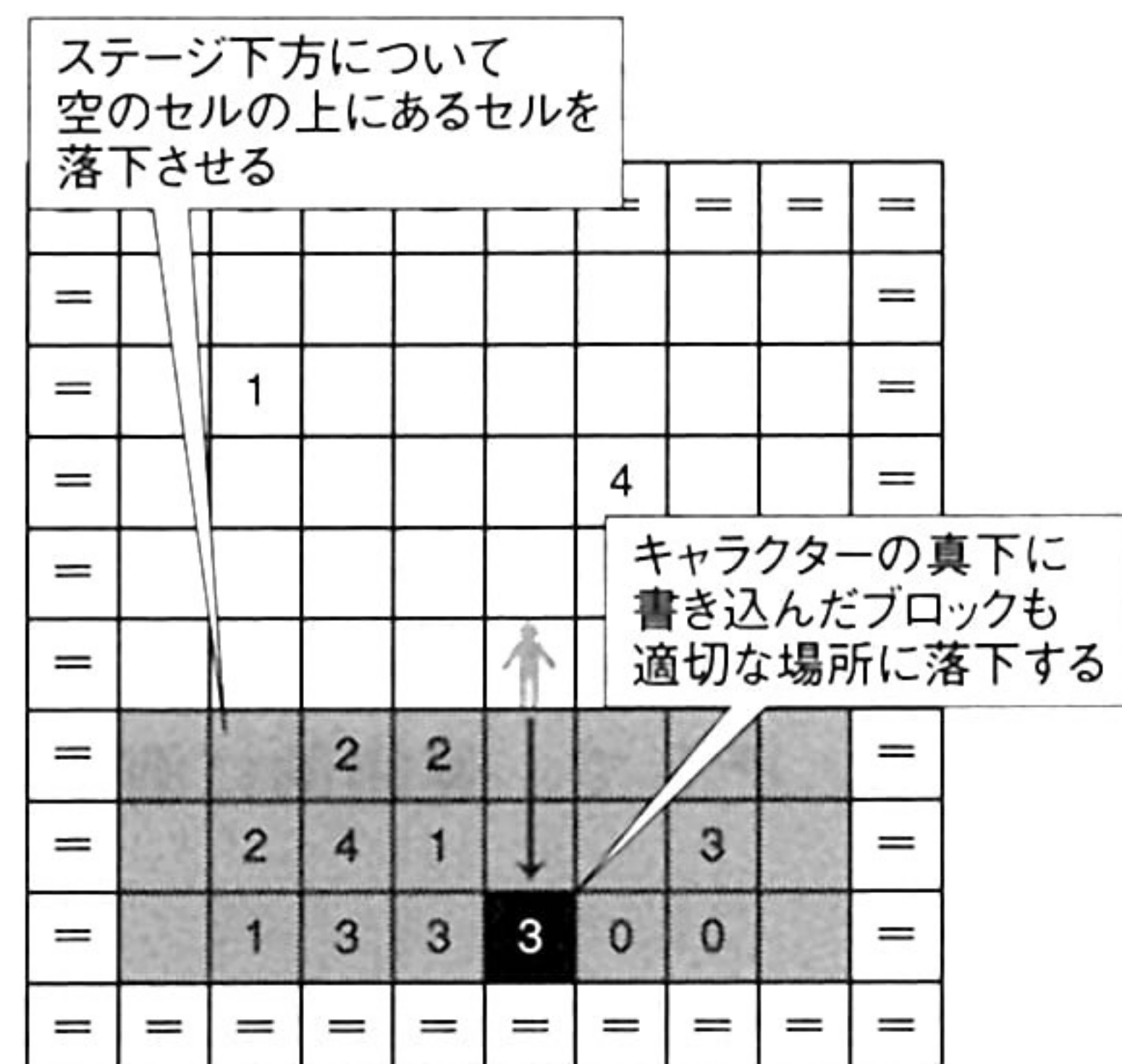
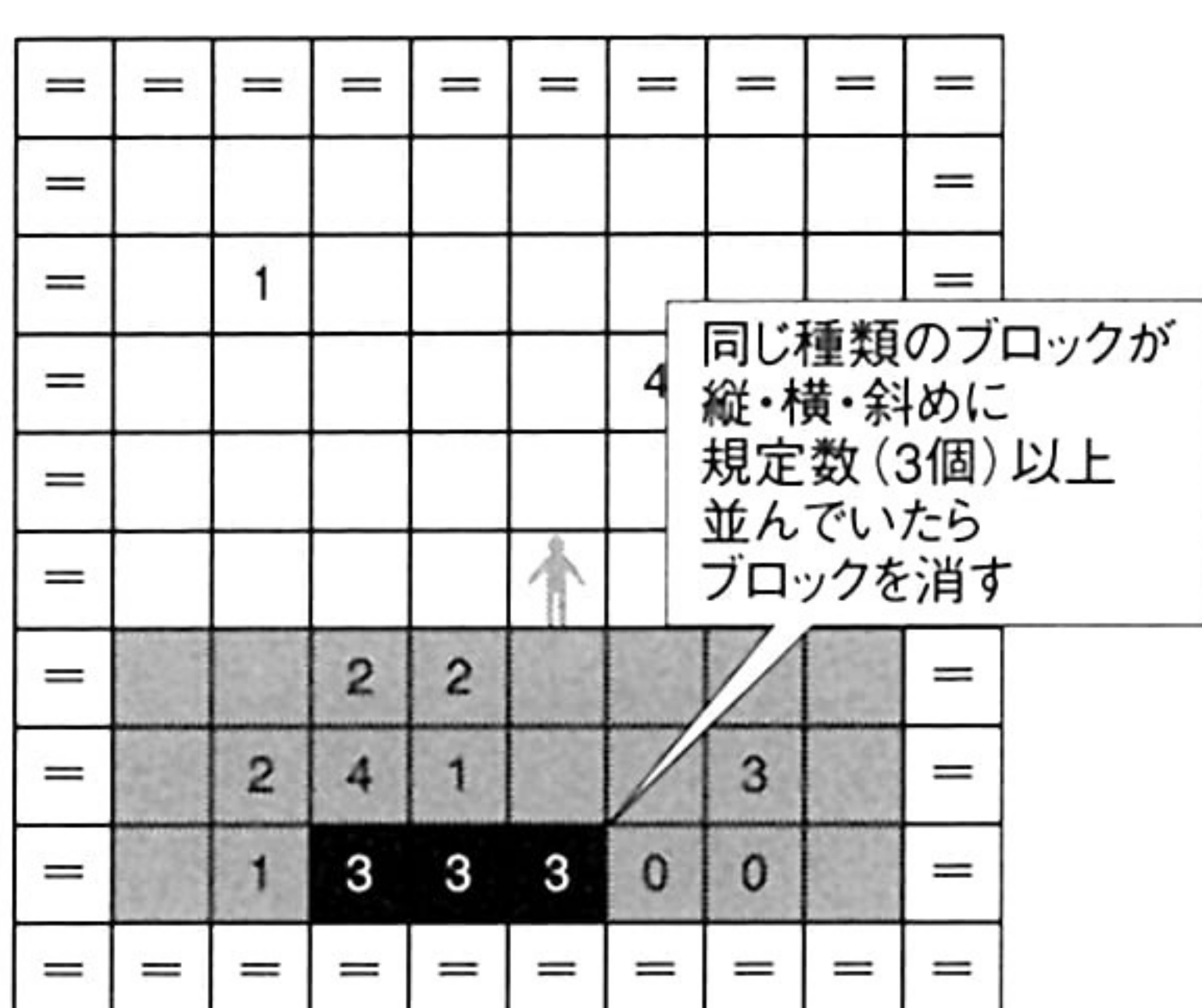
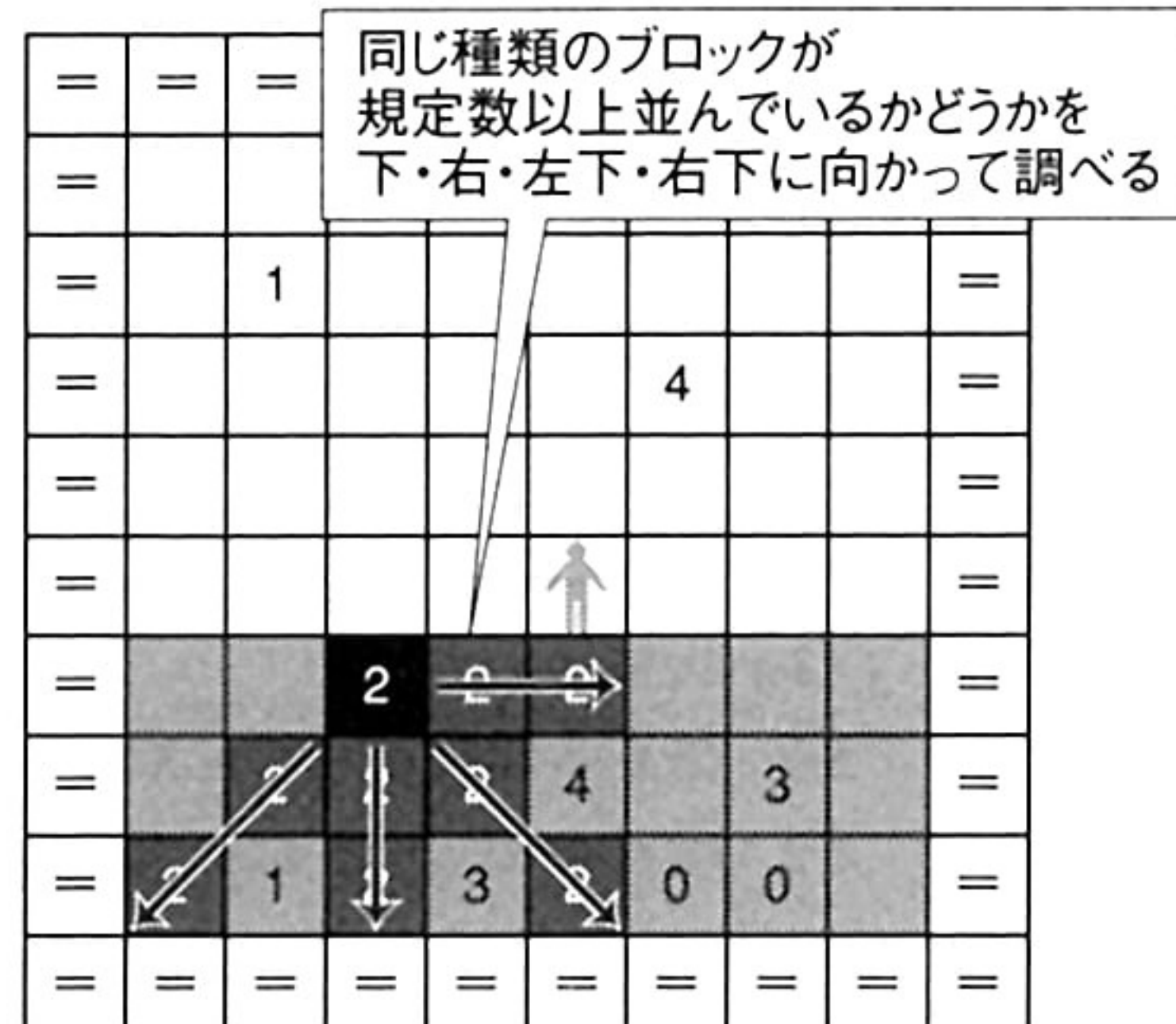



Fig. 4-92　同じ種類のブロックが並んでいるかどうかを調べる

同じ種類のブロックが縦・横・斜めに規定数（3個）以上並んでいたらブロックを消す

Fig. 4-93　ブロックの並びを調べる方法

同じ種類のブロックが規定数以上並んでいるかどうかを下・右・左下・右下に向かって調べる

縦・横・斜めのいずれかに、規定数（3個）以上の同じ種類のブロックが並んでいたら、ブロックを消します。

同じ種類のブロックが並んでいるかどうかを調べる方法は、「縦横斜めに揃える」（→p. 82）と同じです。ステージ内のブロックについて、下・右・左下・右下に向かって、規定数以上の同じブロックが並んでいるかどうかを調べます（Fig. 4-93）。上・左・右上・左上については、それぞれ下・右・左下・右下を調べれば同時に調べられるので、調べる必要はありません。

ブロックが消えたら、空のセルの上にあるセルを下に移動します。このとき、再び同じ種類のブロックが並ぶと、連鎖的に消えます。

プログラム

List 4-10は落ちてくるブロックを拾って積むプログラムです。ステージの移動処理を掲載しました。

移動処理は入力状態・落下状態・消去状態に分かれています。入力状態では、レバー入力に応じて、キャラクターを左右に動かします。ブロックを持っているときにボタンを押したら、キャラクターの真下にブロックを配置して、落下状態に移行します。

入力状態では、一定時間ごとにステージ上方のブロックを落下させる処理も行います。ブロックを落としたときに、真下にキャラクターがいたら、キャラクターはブロックを拾います。

落下状態では、ステージ下方のセルについて、空のセルの上にあるセルを落下させます。次に、同じ種類のブロックが並んでいるかどうかを調べて、規定数以上が縦・横・斜めに並んでいたら、ブロックを消えるブロックにします。そして、消去状態に移行します。

消去状態では、一定時間が経過するのを待ってから、消えるブロックのセルを空にして、完全に消します。そして、連鎖的に消す処理を行うために、再び落下状態に移行します。

List 4-10 落ちてくるブロックを拾って積む（CCaughtAndPiledBlockStageクラス）

```
// ステージの移動処理
bool CCaughtAndPiledBlockStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 入力状態
    if (State==0) {

        // レバー入力に応じて、キャラクターを左右に動かす
        if (!PrevLever) {
            if (is->Left && CX>1) CX--; else
            if (is->Right && CX<xs-2) CX++;
        }
        PrevLever=is->Left||is->Right;
```

```
    // ブロックを持っているときにボタンを押したら、
    // キャラクターの真下にブロックを積む
    if (!PrevButton && is->Button[0] && Block!=' ') {

        // キャラクターの真下のセルが空ならば、
        // ブロックを配置して、
        // 落下状態に移行する
        if (Cell->Get(CX, ys-4)==' ') {
            Cell->Set(CX, ys-4, Block);
            Block=' ';
            State=1;
        }
    }
    PrevButton=is->Button[0];

    // 一定時間ごとにブロックを落下させる
    DropTime++;
    if (DropTime==60) {
        DropTime=0;

        // キャラクターがブロックを持っておらず、
        // 真上にブロックがあるときには、
        // ブロックを拾う
        if (Block==' ' && Cell->Get(CX, CY-1)!=' ') {
            Block=Cell->Get(CX, CY-1);
        }

        // ステージ内のセルを1段ずつ下に移動する
        for (int y=CY-1; y>2; y--) {
            for (int x=1; x<xs-1; x++) {
                Cell->Set(x, y, Cell->Get(x, y-1));
            }
        }

        // 最上段のセルは空にする
        for (int x=1; x<xs-1; x++) {
            Cell->Set(x, 2, ' ');
        }

        // 最上段のセルに新しいブロックを1個書き込む
        // 左右の位置とブロックの種類はランダムに決める
        Cell->Set(
            Rand.Int31()%(xs-2)+1, 2,
            '0'+Rand.Int31()%
                CAUGHT_AND_PILED_BLOCK_COUNT);
    }
}

// 落下状態
```

```
if (State==1) {

    // ステージ下方のセルから、空のセルを探す
    for (int x=1; x<xs-1; x++) {
        for (int y=CY+1; y<ys-1; y++) {

            // 空のセルが見つかったら、上にあるセルを落とす
            if (Cell->Get(x, y)==' ') {

                // 上にあるセルを1段ずつ下に移動する
                for (int i=y; i>CY+1; i--) {
                    Cell->Set(x, i, Cell->Get(x, i-1));
                }

                // 最上段のセルは空にする
                Cell->Set(x, CY+1, ' ');
            }
        }
    }

    // ブロックが消えなかったら、入力状態に移行する
    State=0;

    // ステージ下方のセルについて、
    // 同じ種類のブロックが並んでいるかどうかを調べる
    for (int y=CY+1; y<ys-1; y++) {
        for (int x=1; x<xs-1; x++) {

            // 空のセルは調べない
            char c=Cell->Get(x, y)&0x7f;
            if (c==' ') continue;

            // ブロックの並びを調べる方向
            // （下、右、左下、右下）
            static const int
                vx[]={0, 1, -1, 1},
                vy[]={1, 0, 1, 1};

            // 各方向について、
            // 同じ種類のブロックが並んでいるかどうかを調べる
            for (int v=0; v<4; v++) {

                // 同じ種類のブロックが並んでいる数を調べる
                int count=0;
                for (
                    int i=x, j=y;
                    1<=i && i<xs-1 && CY+1<=j && j<ys-1;
                    i+=vx[v], j+=vy[v], count++
                ) {
                    if (c!=(Cell->Get(i, j)&0x7f)) break;
```

```
                    }

                    // 規定数以上のブロックが並んでいたら、
                    // 消えるブロックとしてマークする
                    if (count>=CAUGHT_AND_PILED_BLOCK_ERASE) {
                        for (
                            int i=x, j=y, k=0;
                            k<count;
                            i+=vx[v], j+=vy[v], k++
                        ) {
                            Cell->Set(i, j, Cell->Get(i, j)|0x80);
                        }

                        // 消去状態に移行する
                        Time=0;
                        State=2;
                    }
                }
            }
        }
    }

    // 消去状態
    if (State==2) {

        // 一定時間が経過したら、
        // 消えるブロックを完全に消す
        Time++;
        if (Time==30) {

            // 消えるブロックを探す
            for (int y=CY+1; y<ys-1; y++) {
                for (int x=1; x<xs-1; x++) {

                    // 消えるブロックが見つかったら、
                    // セルを空にする
                    if (Cell->Get(x, y)&0x80) {
                        Cell->Set(x, y, ' ');
                    }
                }
            }

            // 落下状態に移行する
            State=1;
        }
    }

    return true;
}
```

SAMPLE

「CAUGHT AND PILED BLOCK」は「落ちてくるブロックを拾って積む」のサンプルです。

レバーの左右（カーソルキーの左右）でキャラクターが動きます。落ちてくるブロックの下にキャラクターを移動すると、ブロックを拾うことができます。

ボタン0（Zキー）を押すと、拾ったブロックをキャラクターの真下に積みます。ブロックを積んだときに、同じ種類のブロックが縦・横・斜めのいずれかに規定数（3個）以上並ぶと、ブロックを消すことができます。

ブロックが消えると、上にあったブロックが下に落ちます。このときに同じ種類のブロックが並ぶと、連鎖的に消えます。

CAUGHT AND PILED BLOCK → p. 388

床をマークしてブロックを消す

立体的なステージで、カーソルを動かして床をマークし、ブロックを消すアクションです。マークした床の上にブロックが載ったときに、タイミングよくボタンを押すと、ブロックを床に沈めて消すことができます。

ステージの奥から手前に向かって、多数のブロックが迫ってきます（Fig. 4-94）。図で黒い球で示したのはカーソルです。レバーの上下左右で、カーソルを動かすことができます。

ボタン0（Zキー）を押すと、カーソルの真下にある床をマークすることができます（Fig. 4-95）。すでに床がマークされているときには、マークを解除します。一度に複数の床をマークすることも可能です。ブロックがマークされた床に載ったときに、タイミングよくボタン1（Xキー）を押すと、ブロックを床に沈めて消すことができます（Fig. 4-96）。ブロックを次々に消していくことがゲームの目的です。

床をマークしてブロックを消すゲームには『I.Q』があります。このゲームでは、画面奥から

Fig. 4-94 迫ってくるブロック

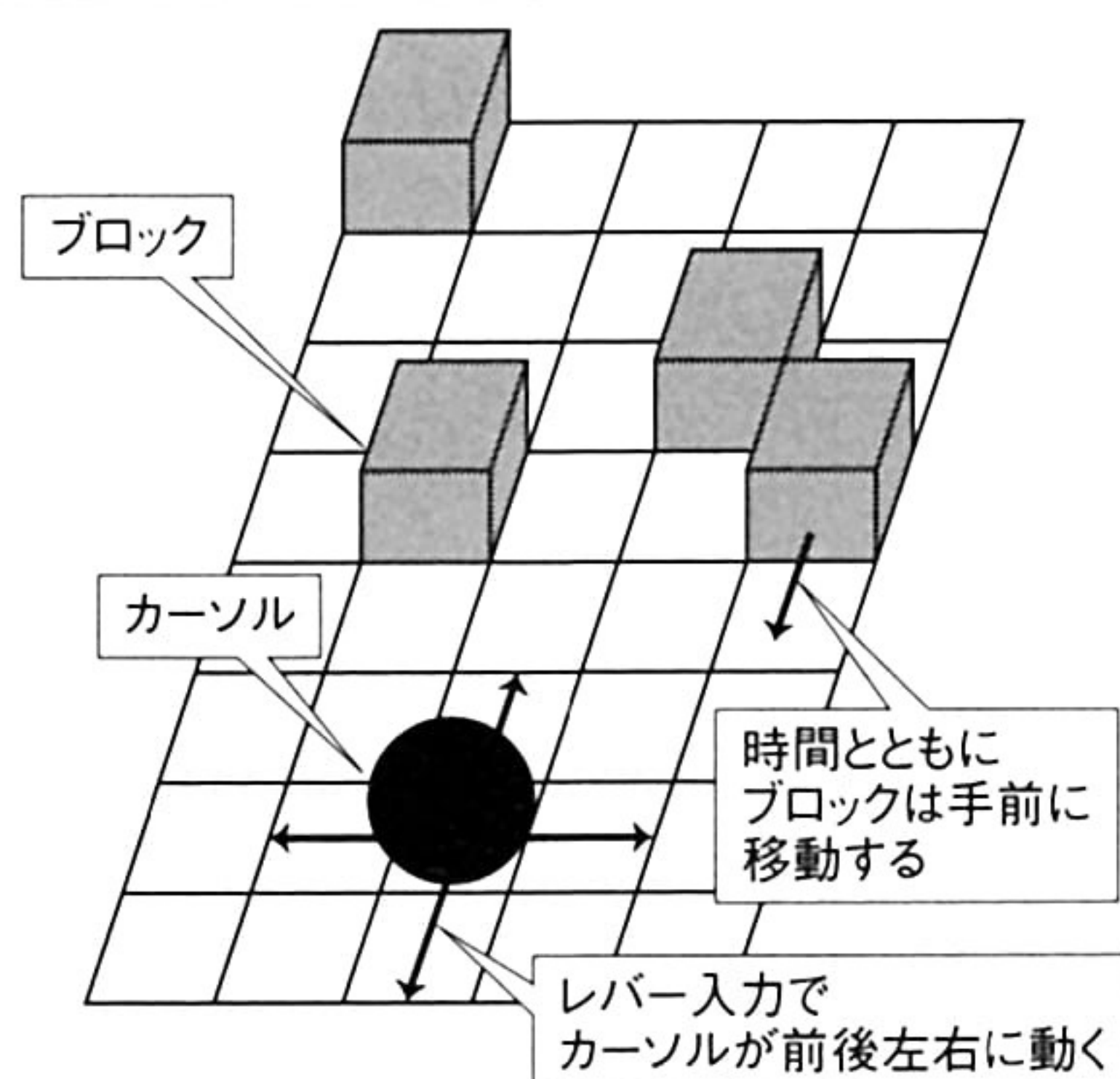



Fig. 4-95 床をマークする

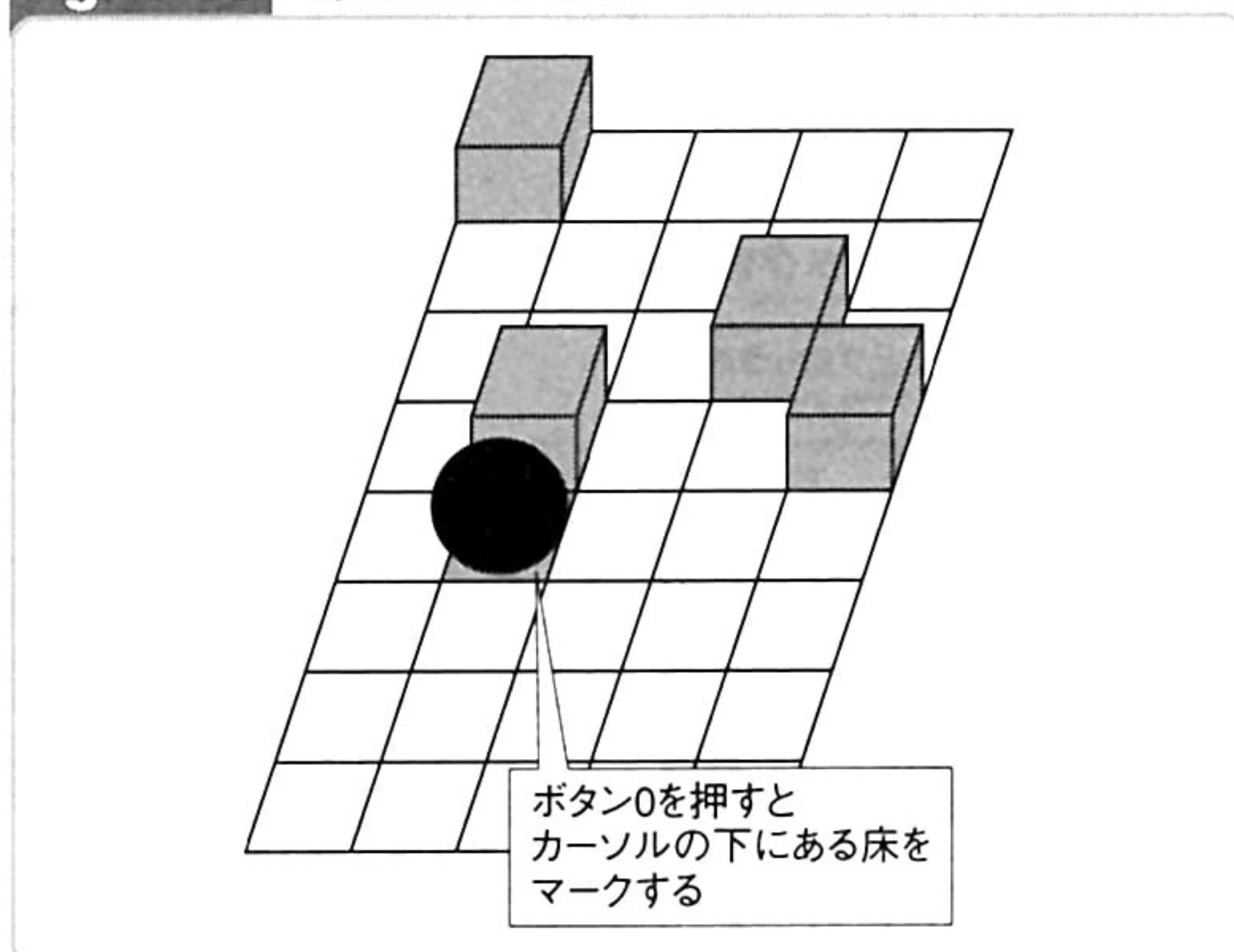
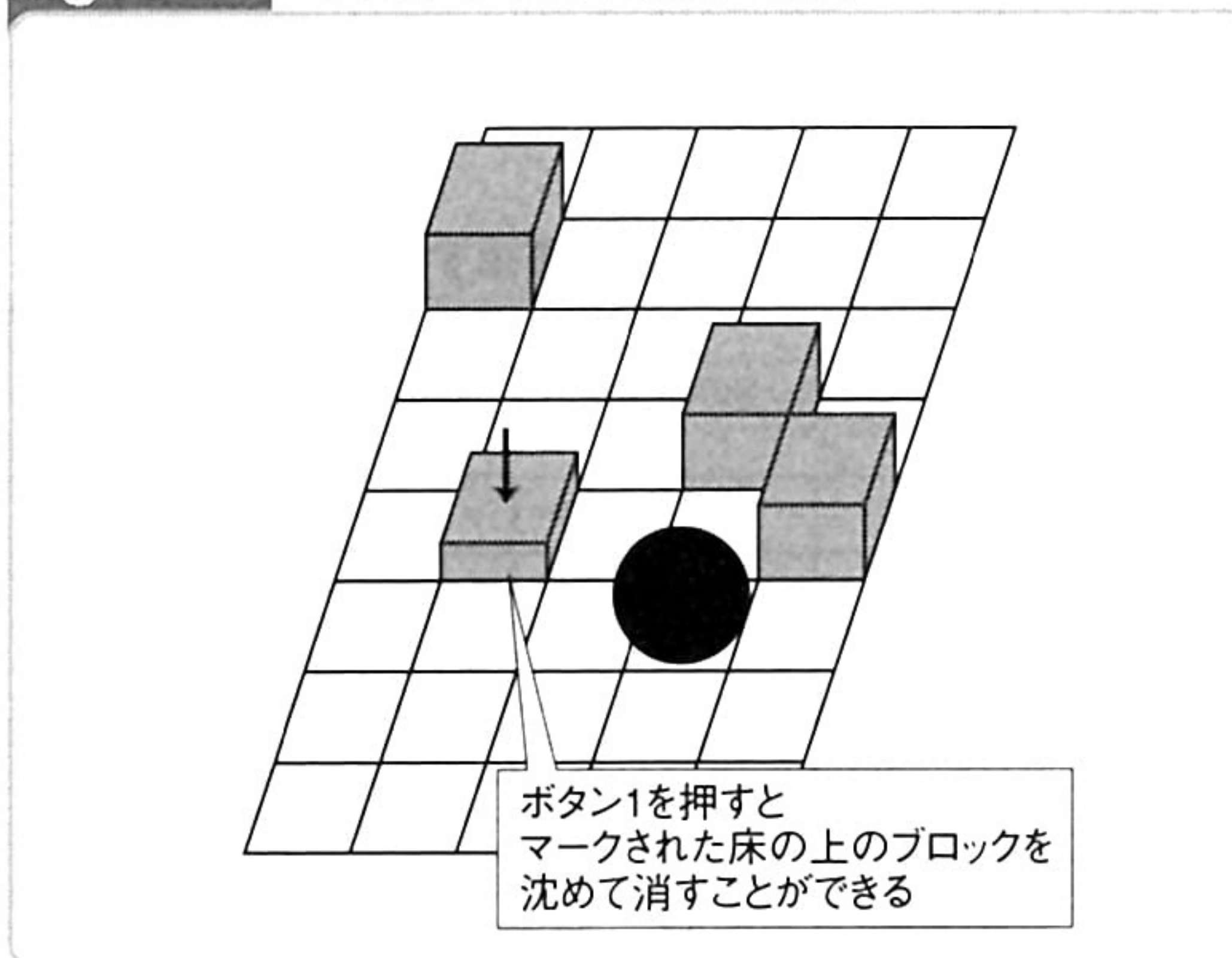



Fig. 4-96 ブロックを床に沈めて消す

多数のブロックが転がってきます。キャラクターを動かして床をマークし、タイミングよくボタンを押すと、床の上のブロックを沈めて消すことができます。

ブロックがステージの手前まで到達すると、ペナルティとして、ステージが狭くなります。キャラクターがステージから落ちてしまうと、ミスになります。

このゲームが面白いのは、キャラクターがブロックに押しつぶされることです。ブロックを巧みに避けながら、床をマークしなければなりません。さらに、沈めてはいけないブロックも出現するため、本当に必要な床だけを選んでマークする必要があります。アクション性とパズル性が適度にミックスされた、スリリングなゲームです。

アルゴリズム

床をマークしてブロックを消すアクションを実現するには、ステージを3次元のセルで表現します(Fig. 4-97)。3次元とはいえ、カーソルとブロックがある層と、床がある層の2層だけです。図では各層のセルを平面で示しました。ブロックは「#」で、床は「=」で表すことにします。

ブロックの層のセルは、一定時間ごとに奥から手前に移動します(Fig. 4-98)。これで、ブロックが奥から手前に少しずつ迫ってきます。タイマーを使って、ブロックが転がる演出をすると、より見た目が楽しくなるでしょう。

ボタン0を押したら、カーソルの真下にある床を調べます。マークされていない床(=)ならば、マークされた床に変化させます(Fig. 4-99)。マークされた床は「-」で表すことにしました。逆にマークされた床ならば、マークされていない床に変化させます。

ボタン1を押したら、マークされた床に載っているブロックを探します(Fig. 4-100)。マークされた床があり、前後左右が同じ位置にブロックがあれば、そのブロックを沈めて消します。本書のサンプルでは、ブロックのセル(#)を消えるブロック(+)に変化させます。

一定時間が経過したら、消えるブロックを空のセルに変えて、完全に消します(Fig. 4-101)。

完全に消すまでには、ブロックが床に沈む演出を行います。床に沈める方法は簡単で、ブロックを少しずつ下方向に移動するだけです。

Fig. 4-97 ステージをセルで表現する

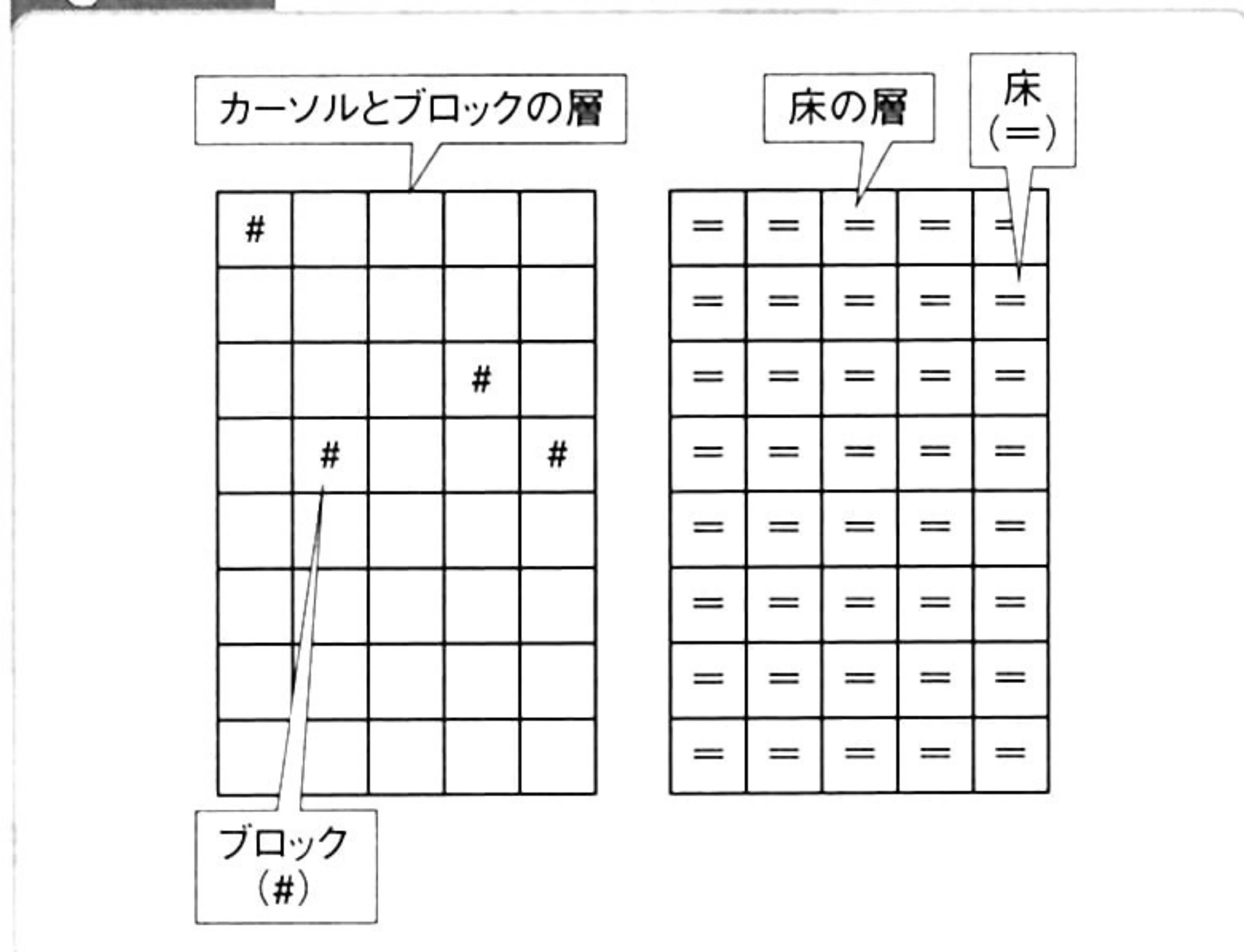



Fig. 4-98 ブロックのセルを奥から手前に移動する

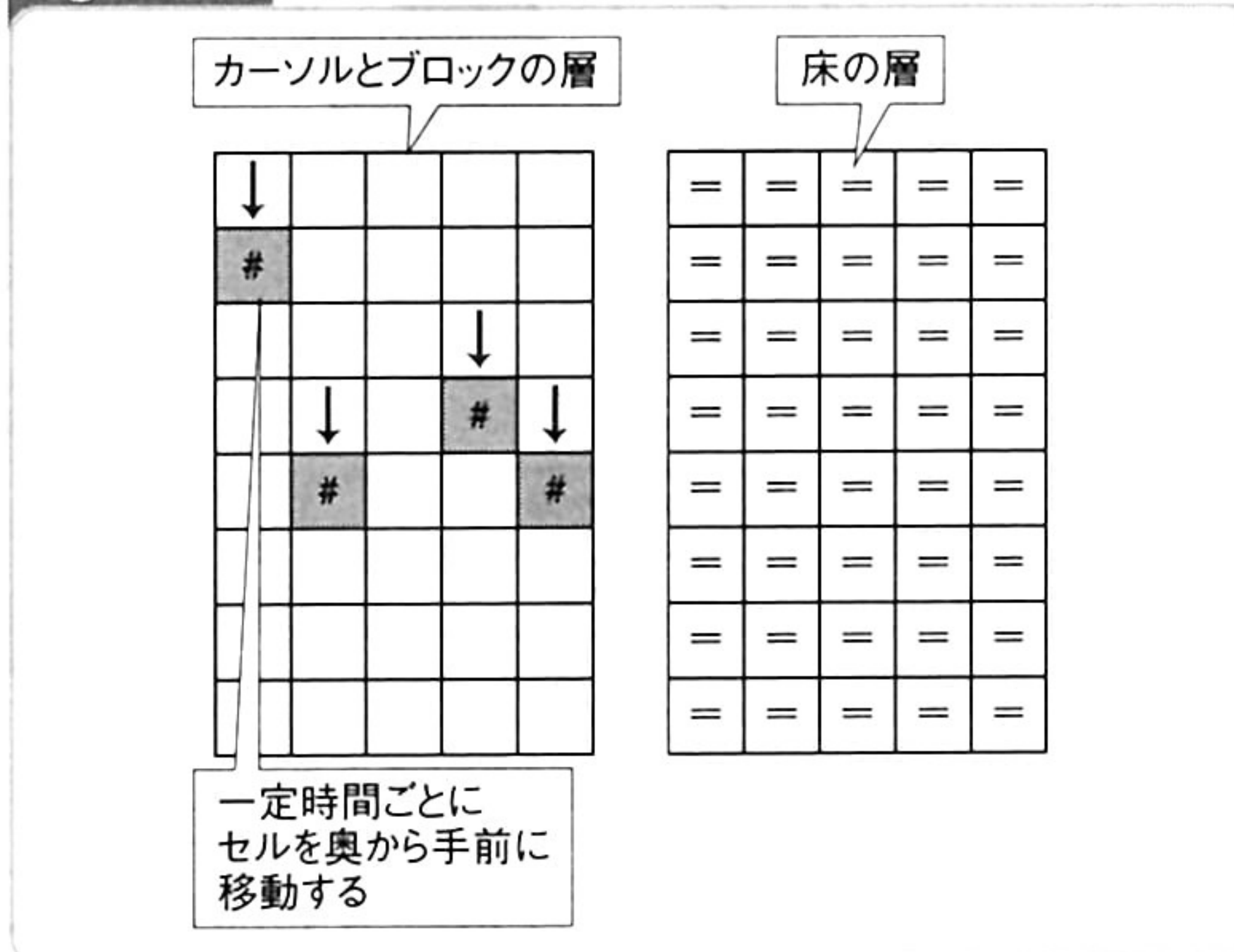



Fig. 4-99 床をマークする

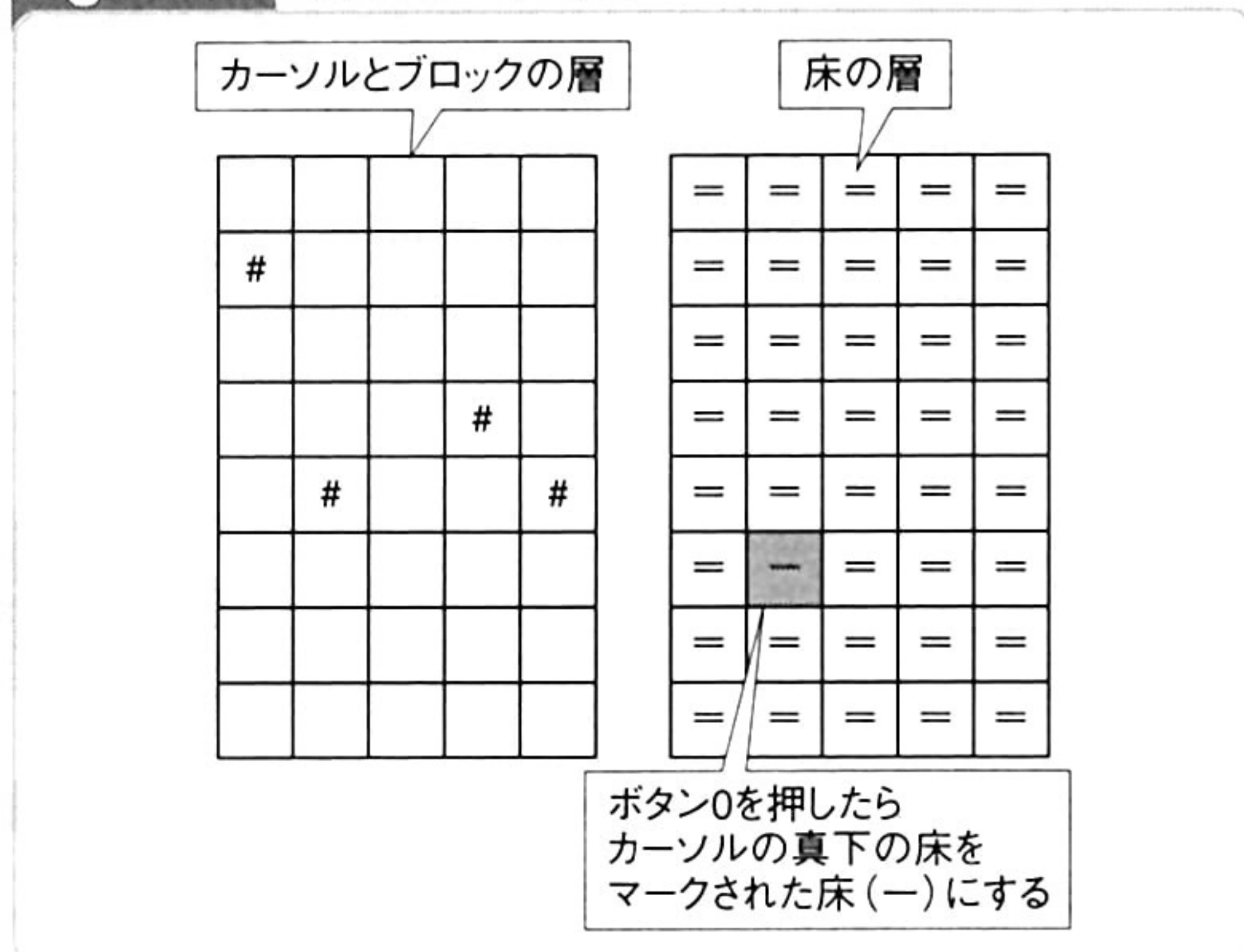



Fig. 4-100 マークされた床の上のブロックを消す

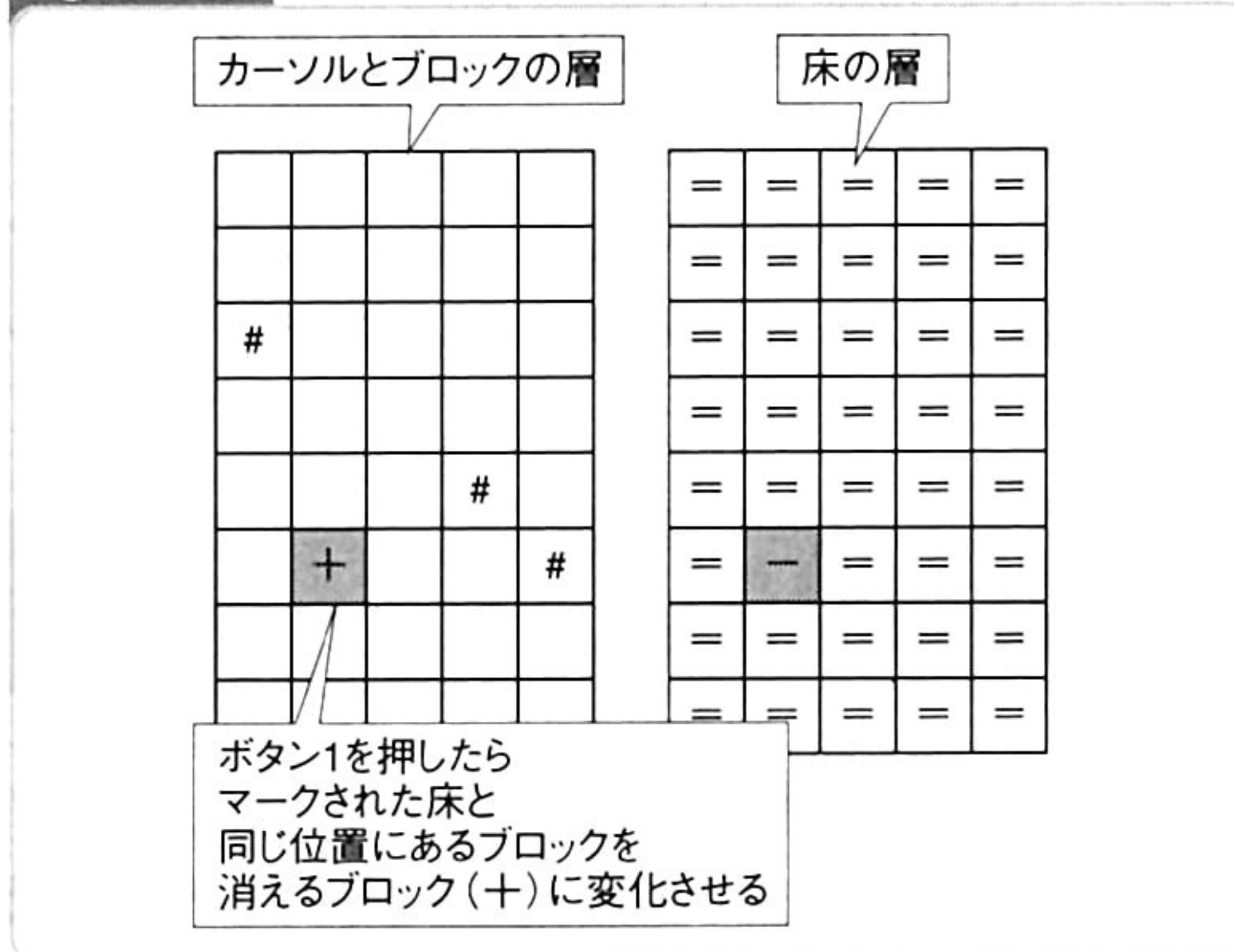
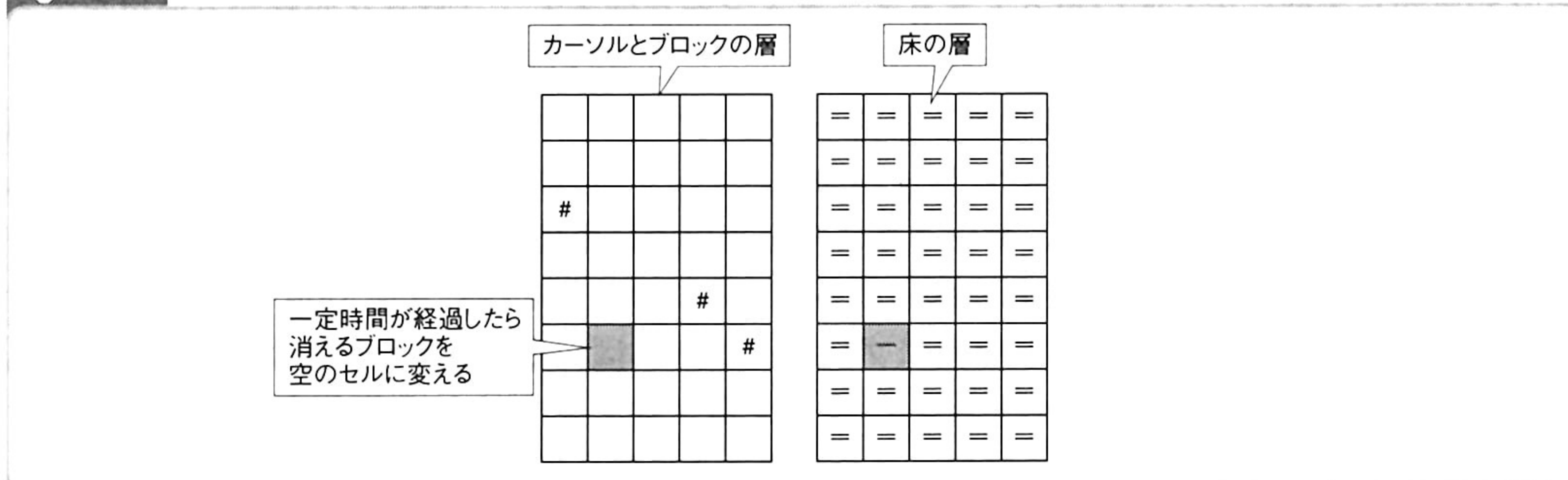



Fig. 4-101 消えるブロックを完全に消す

プログラム

List 4-11は床をマークしてブロックを消すプログラムです。ステージの移動処理を掲載しました。

移動処理は入力状態と消去状態に分かれています。入力状態では、レバー入力に応じて、カーソルを前後左右に動かします。このプログラムでは行いませんが、キューブとカーソルの当たり判定処理を行えば、カーソルがキューブに踏みつぶされるようにすることもできます。

入力状態では、ボタンを押したときの処理も行います。ボタン0を押したら、カーソルの下にある床をマークします。床がすでにマークされていたら、マークを解除します。

ボタン1を押したら、マークされた床に載っているブロックを沈めます。マークされた床の上にあるブロックを探して、消えるブロックに変化させます。そして、消去状態に移行します。

消去状態では、一定時間が経過するのを待ってから、消えるブロックを完全に消去します。消えるまでの間は、タイマーを使ってブロックの描画座標を少しずつ変化させ、ブロックが沈む演出を行います。具体的な処理は、ステージの描画処理（CMarkedAndSunkBlockStageクラスのDraw関数）を参照してください。

入力状態では、ブロックを奥から手前に移動する処理も行います。一定時間ごとにステージ内のセルを手前に移動するとともに、最も奥の部分に、新しいブロックを出現させます。

List 4-11 床をマークしてブロックを消す（CMarkedAndSunkBlockStageクラス）

```
// ステージの移動処理
bool CMarkedAndSunkBlockStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize(), zs=Cell->GetZSize();

    // 入力状態
    if (State==0) {

        // レバー入力に応じて、カーソルを前後左右に動かす
        if (!PrevLever) {
            if (is->Left && CX>0) CX--;
            if (is->Right && CX<xs-1) CX++;
            if (is->Up && CZ<zs-1) CZ++;
            if (is->Down && CZ>0) CZ--;
        }
        PrevLever=is->Left||is->Right||is->Up||is->Down;

        // ボタンを押したときの処理
        if (!PrevButton) {

            // ボタン0を押したら、カーソルの下にある床をマークする
            if (is->Button[0]) {
                switch (Cell->Get(CX, 0, CZ)) {
```

```
                //床がマークされていなければ、マークする
                case '=': Cell->Set(CX, 0, CZ, '-'); break;

                //床がマークされていたら、マークを解除する
                case '-': Cell->Set(CX, 0, CZ, '='); break;
            }
        } else

        //ボタン1を押したら、
        //マークされた床の上にあるブロックを沈めて消す
        if (is->Button[1]) {

            //マークされた床の上にあるブロックを探す
            for (int z=0; z<zs; z++) {
                for (int x=0; x<xs; x++) {

                    //ブロックを見つけたときの処理
                    if (
                        Cell->Get(x, 1, z)=='#' &&
                        Cell->Get(x, 0, z)=='-'
                    ) {
                        //ブロックを消えるブロックにする
                        Cell->Set(x, 1, z, '+');

                        //消去状態に移行する
                        Time=0;
                        State=1;
                    }
                }
            }
        }
    }
    PrevButton=is->Button[0]||is->Button[1];

    //一定時間ごとに、ブロックを手前に移動する
    DropTime++;
    if (DropTime==60) {
        DropTime=0;

        //ステージ内のセルを手前に移動する
        for (int z=0; z<zs-1; z++) {
            for (int x=0; x<xs; x++) {
                Cell->Set(x, 1, z, Cell->Get(x, 1, z+1));
            }
        }

        //最も奥のセルを空にする
        for (int x=0; x<xs; x++) {
            Cell->Set(x, 1, zs-1, ' ');
```

```
            }

            // 最も奥のセルにブロックを1個出現させる
            // 左右方向の位置はランダムに決める
            Cell->Set(Rand.Int31()%xs, 1, zs-1, '#');
        }
    }

    // 消去状態
    if (State==1) {

        // 一定時間が経過したら、消えるブロックを完全に消す
        Time++;
        if (Time==30) {

            // 消えるブロックを探す
            for (int z=0; z<zs; z++) {
                for (int x=0; x<xs; x++) {

                    // 消えるブロックを見つけたら、セルを空にする
                    if (Cell->Get(x, 1, z)=='+') {
                        Cell->Set(x, 1, z, ' ');
                    }
                }
            }

            // 入力状態に移行する
            State=0;
        }
    }
    return true;
}
```

SAMPLE

「MARKED AND SUNK BLOCK」は「床をマークしてブロックを消す」のサンプルです。

黒い球のカーソルは、レバーの上下左右（カーソルキーの上下左右）で前後左右に動きます。ボタン0（Zキー）を押すと、カーソルの真下にある床をマークします。すでにマークしてある床の上でボタンを押したときには、マークを解除します。複数の床を同時にマークすることもできます。

ボタン1（Xキー）を押すと、マークの上にあるブロックが沈んで消えます。複数のマークの上にブロックがあるときには、複数のブロックをまとめて沈めることができます。

MARKED AND SUNK BLOCK → p. 389

まとめ

本章では「ブロック」を使ったさまざまなアクションを紹介しました。ブロックを使ったパズルゲームは数多くあり、それぞれに違ったゲーム性があります。しかし、ブロックを落としたり、同じ種類のブロックを並べたりといった基本的なアクションは、多くのゲームに採用されています。基本的なアクションに対して、何か変わった操作や、楽しい演出などを加えることによって、オリジナリティのあるゲームになります。

というわけで、「ブロックの基本アクションを押さえつつ、オリジナルの要素を盛り込もう！」というのが本章のまとめです。

Stage

05

ボール

Ball

「ボール」も「ブロック」とともに、数多くのパズルゲームで使われている要素です。ボールはブロックと同じように、落としたり、並べたり、つなげたり、撃ったりすることができます。一方で、軌道に沿ってボールが進むとか、撃ったボールが壁で跳ね返るとか、ボールが転がって進むといった、ボールの丸い形状を生かした使い方もあります。

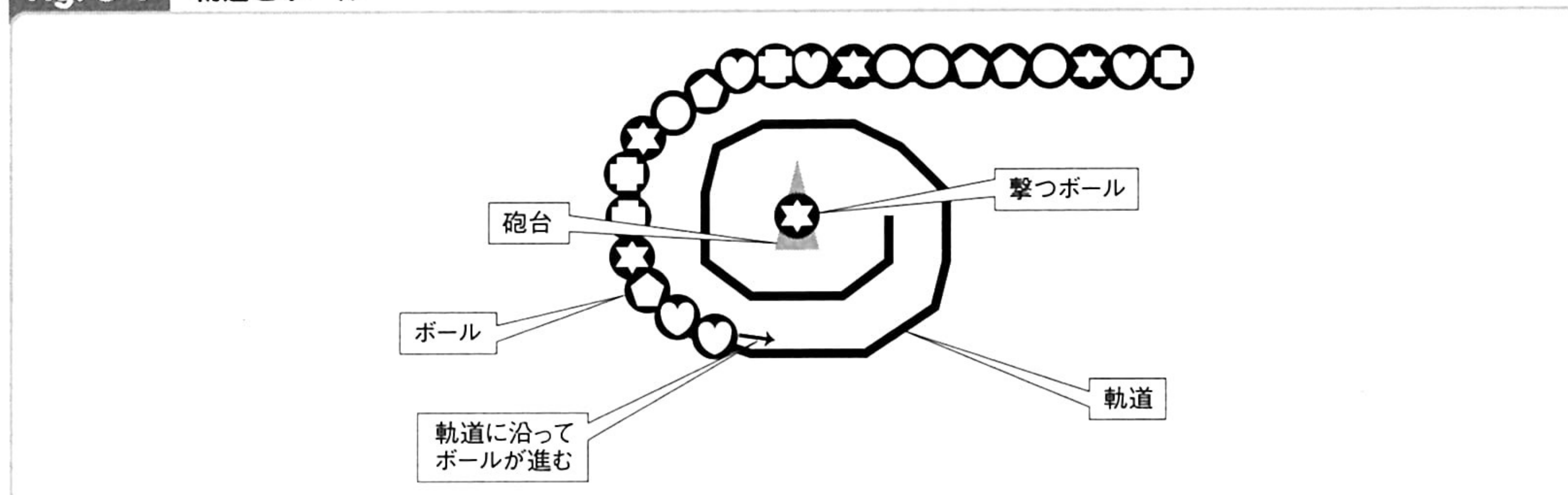
軌道に沿って進むボール

さまざまな形をした軌道上を進むボールです。直線だけではなく、波打っていたり、渦を巻いていたりする軌道に沿って、数多くのボールが滑らかに進みます。

ステージには軌道が配置されています（Fig. 5-1）。本書のサンプルでは、渦巻状の軌道を作成しました。この軌道は少し角張っていますが、必要に応じて、軌道を構成する線分の区画を細かくすることによって、簡単に滑らかにすることができます。

軌道に沿って、多数のボールが少しずつ進んでいきます。軌道の最後までボールが達すると、ゲームオーバーになるゲームもあります。本書のサンプルでは、ミスにはならず、単にボールが消えます。

ステージの中央にあるのは、ボールを撃つための砲台です。砲台を使ってボールを撃ち、軌道上にボールを撃ち込むことができます。

Fig. 5-1 軌道とボール

軌道に沿って進むボールを採用したゲームには『パズループ』があります。このゲームでは、軌道上を進んでくるボールに対して、砲台を使って新しいボールを撃ち込みます。軌道上で同じ種類のボールが規定数（3個）以上並ぶと、ボールを消すことができます。ボールが軌道の末端に到達しないように、次々にボールを消していくことがゲームの目的です。

アルゴリズム

軌道に沿って進むボールを実現するためのポイントは、さまざまな形をした軌道を表現する方法と、軌道の形に沿ってボールを進める方法です。まず、軌道を表現する方法について考えましょう。

軌道を表現する

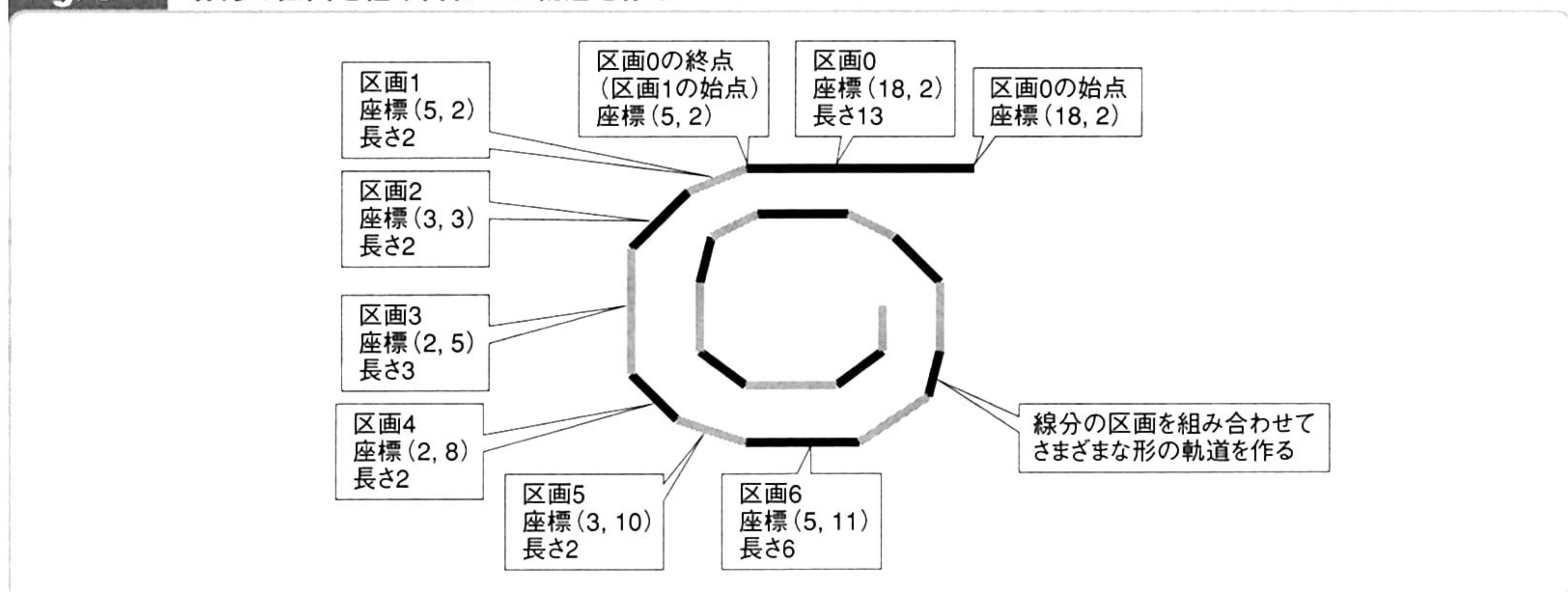
本書では、線分の区画を組み合わせて（つまりは線をつなげて）、さまざまな形の軌道を作ることにします（Fig. 5-2）。軌道のスタート地点に近い方から、区画0・区画1・区画2…のように番号を振ります。

図では軌道が角張っていますが、区画の数を増やせば、いくらでも滑らかな軌道を作ることができます。軌道を滑らかにしても、データを作る手間が増えるだけで、処理の方法はまったく変わりません。

それぞれの区画について、座標と長さのデータを用意します。線分には始点と終点がありますが、終点は次の区画の始点と共通なので、始点の座標があれば十分です。座標と長さの単位は、ボールの直径に相当する長さを1にしておくと便利です。

なお、図では簡単のため、線分の長さを大まかな整数値にしています。実際のプログラムでは、線分の座標から、正確な長さの値を計算しています。

Fig. 5-2 線分の区画を組み合わせて軌道を作る

区画1
座標（5, 2）
長さ2
区画0の終点
（区画1の始点）
座標（5, 2）
区画0
座標（18, 2）
長さ13
区画0の始点
座標（18, 2）
区画2
座標（3, 3）
長さ2
区画3
座標（2, 5）
長さ3
区画4
座標（2, 8）
長さ2
区画5
座標（3, 10）
長さ2
区画6
座標（5, 11）
長さ6
線分の区画を組み合わせて
さまざまな形の軌道を作る

ボールの位置を確認する

次に、軌道に沿ってボールを進める方法を考えます。本書では、軌道上のボールの位置を、軌道のスタート地点から進んだ距離で表すことにしました（Fig. 5-3）。距離の単位は、ボールの直径を1としています。例えば距離が10ならば、軌道のスタートからボール10個分進んだということです。

区画のデータとボールの位置を使うと、ボールがどの区画上にあるのかがわかります（Fig. 5-4）。ボールの位置から、軌道の区画の長さを、スタートから始めて順番に引いていきます。ボールの位置が、軌道の区画の長さよりも小さくなったら、ボールはその区画上にあるということです。

例えば、ボールの位置を「23」とし、スタートからの区画の長さを、それぞれ「13, 2, 2, 3, 2,

2, 6」とします。ボールの位置から、区画の長さを順番に引いていくと、「10, 8, 6, 3, 1」となり、位置は区画5（6番目の区画）の長さよりも小さくなります。この場合、ボールは区画5の上にある、ということになります。

Fig. 5-3 ボールの位置を軌道上の距離で表す

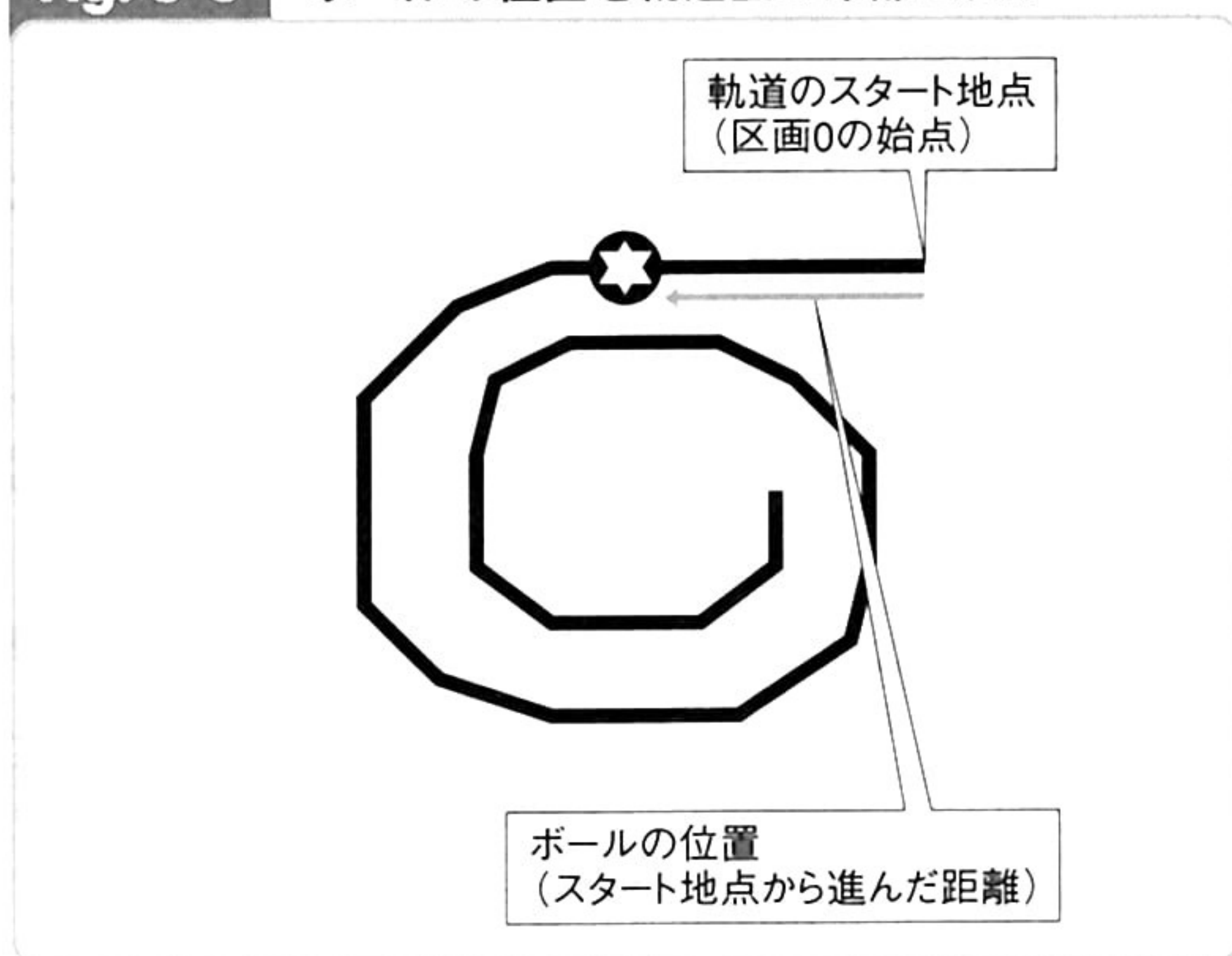
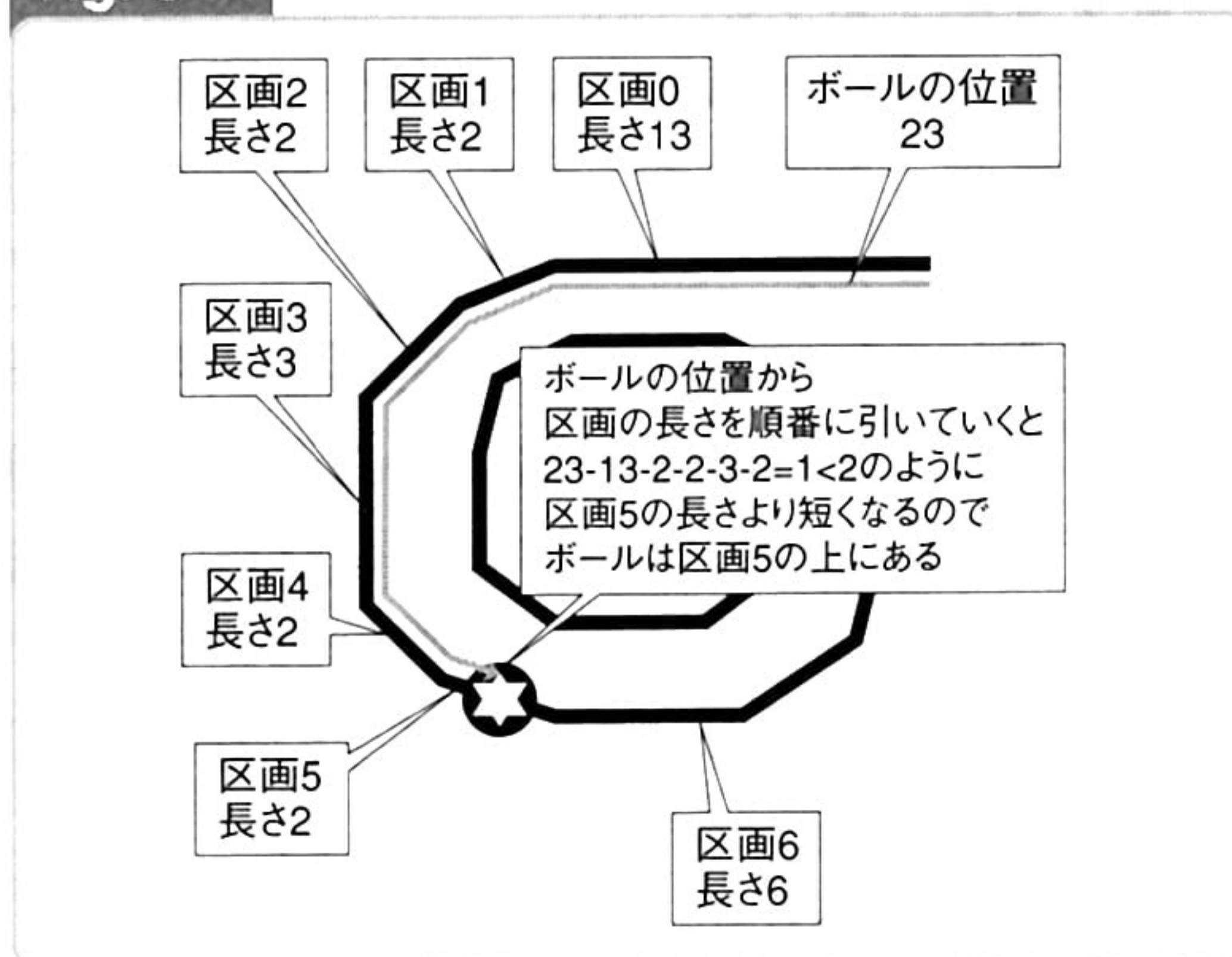



Fig. 5-4 ボールがどの区画上にあるのかを求める

ボールの座標を計算する

ボールがある区画がわかったら、ボールの位置と区画の座標を使って、ボールの座標を計算します（Fig. 5-5）。区画の始点からボールまでの距離（ボールの位置から区画の長さを順番に引いた残り）をpos、ボールがある区画の座標を（x0, y0）、次の区画の座標を（x1, y1）、区画の長さをlengthとすると、ボールの座標（x, y）は、

x=x0+（x1-x0）×pos÷length
y=y0+（y1-y0）×pos÷length

となります。これは、ボールの区画内の相対位置を使って、区画の両端の座標を線形補間する計算です。

例えば、ボールが区画5にあるとします。区画の始点からボールまでの距離を1、区画5の座標を（3, 10）、区画6の座標を（5, 11）、区画5の長さを2とすると、ボールの座標（x, y）は、

x=3+（5-3）×1÷2=4
y=10+（11-10）×1÷2=10.5

となります。

これでボールの軌道上の位置から、ボールの座標を求めることができました。求めた座標は、ボールの描画や、ボール同士の当たり判定処理に使います。あとはボールの位置を時間とともに少しずつ増加させれば、ボールは軌道に沿って進んでいきます。

ボールの位置について、もう1つ考える必要があるのは、ボールとボールの間隔です。ボール同士が近づきすぎたときには、離す必要があります。逆に離れすぎたときには、近づける必

要があります。

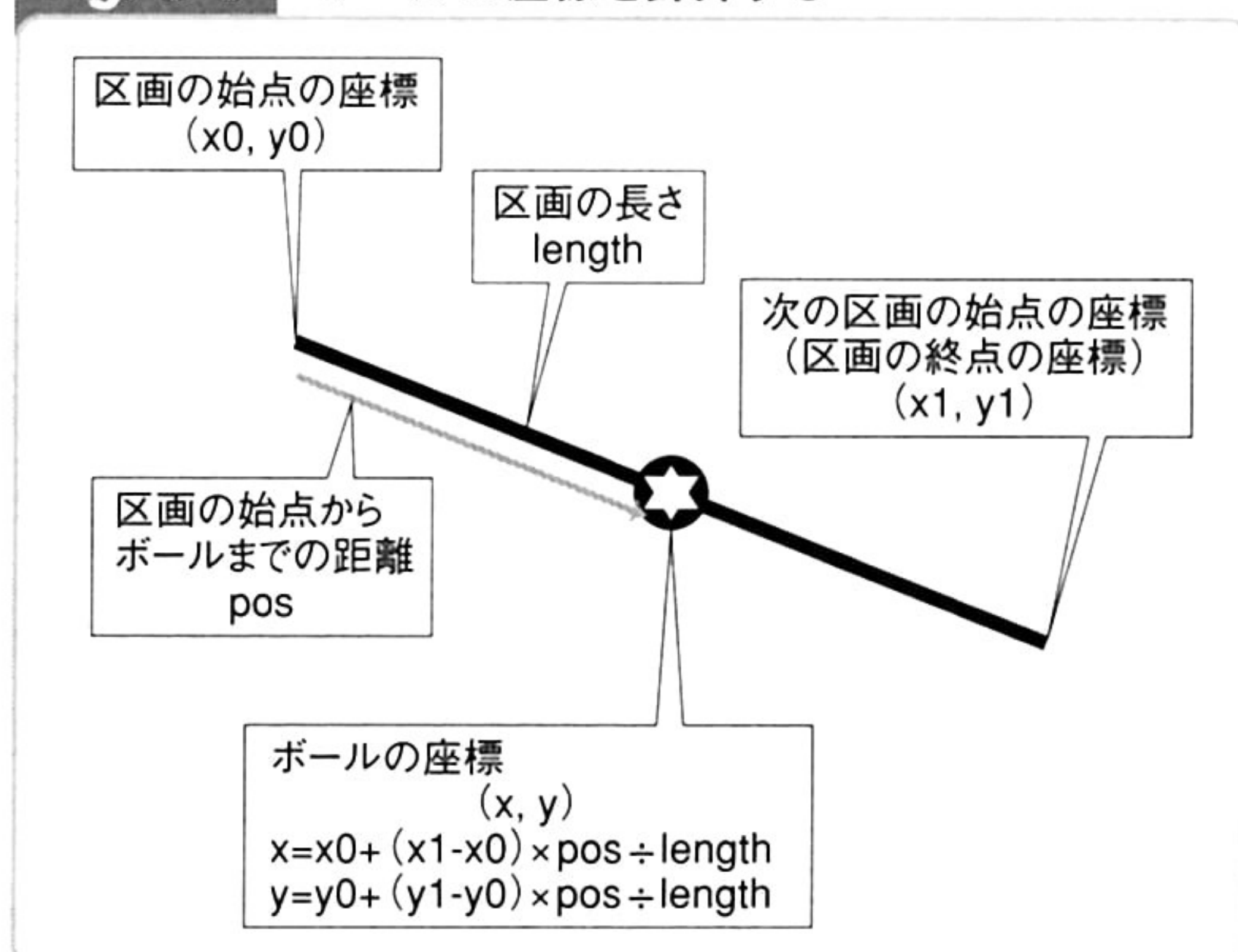
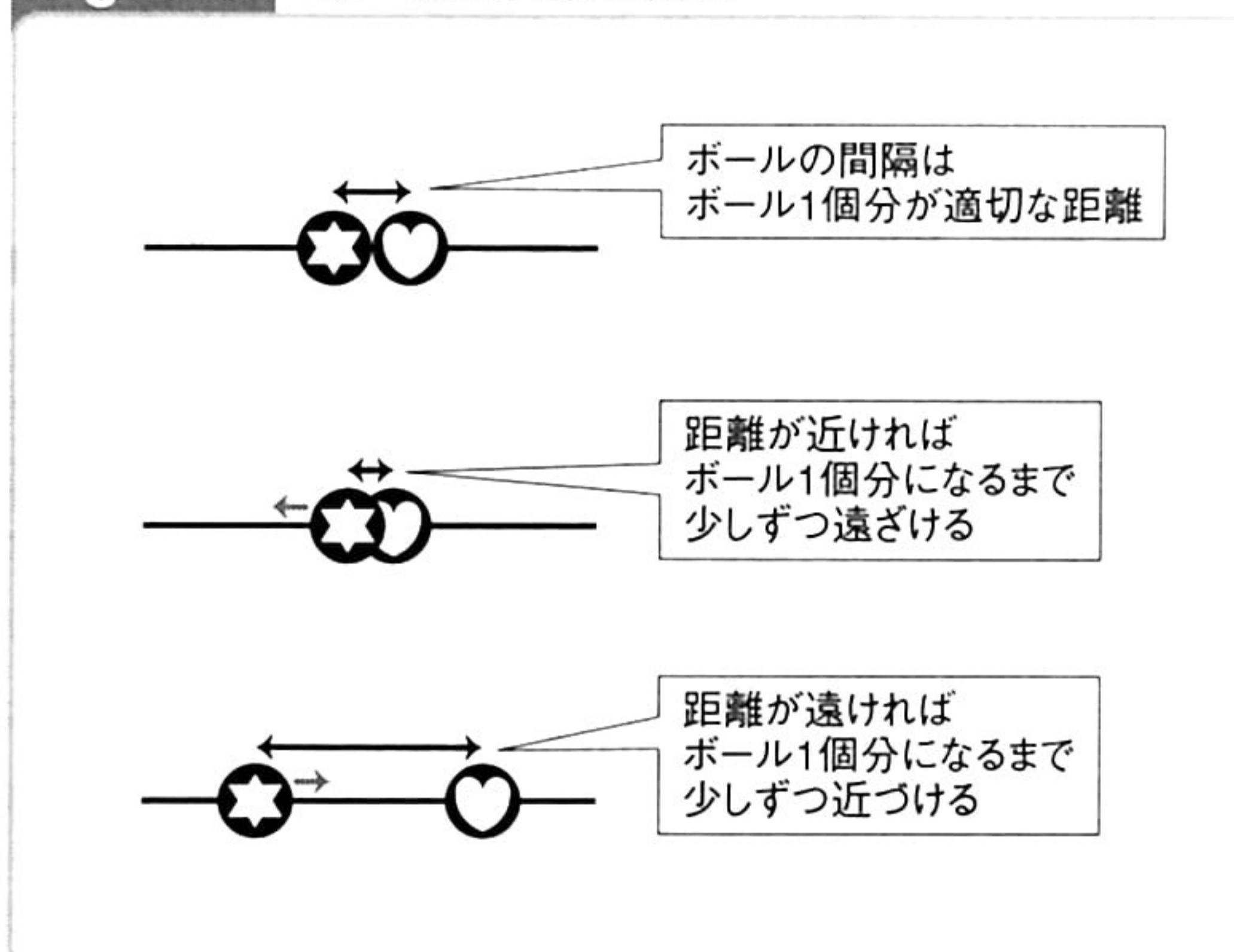
ボール同士の適切な距離は、ボール1個分です。ボール1個分だけ離れていると、ボール同士がちょうど接した状態になります（Fig. 5-6）。1個分よりも近づいていたら、1個分になるまで、少しずつ遠ざけます。逆に1個分よりも離れていたら、1個分になるまで、少しずつ近づけます。

ボールを遠ざけたり近づけたりするときには、両方のボールを動かす方法と、片方のボールだけ動かす方法とがあります。本書のサンプルでは、前方のボール（軌道のスタート地点から遠いボール）だけを動かすことにしています。

Fig. 5-5 ボールの座標を計算する

区画の始点の座標
（x0, y0）
区画の長さ
length
次の区画の始点の座標
（区画の終点の座標）
（x1, y1）
区画の始点から
ボールまでの距離
pos
ボールの座標
（x, y）
x=x0+（x1-x0）×pos÷length
y=y0+（y1-y0）×pos÷length

Fig. 5-6 ボールの間隔の調整

ボールの間隔は
ボール1個分が適切な距離
距離が近ければ
ボール1個分になるまで
少しずつ遠ざける
距離が遠ければ
ボール1個分になるまで
少しずつ近づける

プログラム

List 5-1は軌道に沿って進むボールのプログラムです。ボールと軌道の構造体、軌道のデータ、ステージのコンストラクタ・移動処理・描画処理、ボールの座標を計算する処理を掲載しました。

ボールの構造体（BALL_ON_RAIL_BALL）は、個々のボールに関する情報です。ボールの種類・位置・座標・不透明度を記録します。不透明度はボールを消すときに使います。詳細は「軌道上に並んだボールを消す」（→p. 297）で解説します。

軌道の構造体（BALL_ON_RAIL_RAIL）は、軌道の区画に関する情報です。区画ごとに、座標と長さを記録します。

軌道のデータ（配列RailOnRailRail）は、軌道の区画に関するデータです。区画ごとに座標のデータを用意しました。長さは仮に0にしておき、正確な値はコンストラクタ（CBallOnRail Stage）で計算します。

移動処理（Move関数）では、軌道に沿ってボールを進めます。末尾の（スタートに一番近い）ボールを少し進ませてから、残りのボールの位置を調整します。位置を調整すると、末尾のボールに残りのボールが押し出されて、すべてのボールが進みます。

ボールを進めたら、ボールの座標を計算する処理（SetBallXY関数）を呼び出して、ボールの

位置から座標を計算します。ボールの位置から区画の長さを順番に引くことによって、ボールがどの区画上にあるのかを判定します。次に、区画の両端の座標を線形補間して、ボールの座標を求めます。

ボールがどの区画上にもないときには、ボールを削除します。ボールを削除する処理(EraseBall関数)の詳細は、「軌道上に並んだボールを消す」(→p. 297)で解説します。

描画処理(Move関数)では、軌道とボールを描画します。軌道については太い線分で描画し、ボールについては種類に応じたグラフィックを描画します。

太い線分を描画するために、線分の両端の座標を、線分に対して垂直な方向に少しずつずらして、4つの座標を求めています。これらの座標を使って矩形を描画すれば、太い線分を描くことができます。

List 5-1 軌道に沿って進むボール(CBallOnRailStageクラス)

```
// ボールの構造体
typedef struct {

    // 種類
    int Type;

    // 軌道上の位置、座標、不透明度(アルファ値)
    float Pos, X, Y, Alpha;

} BALL_ON_RAIL_BALL;

// 軌道の構造体
typedef struct {

    // 座標、長さ
    float X, Y, Length;

} BALL_ON_RAIL_RAIL;

// 軌道のデータ(座標、長さ)
BALL_ON_RAIL_RAIL BallOnRailRail[]={
    {18, 2, 0},
    {5, 2, 0},
    {3, 3, 0},
    {2, 5, 0},
    {2, 8, 0},
    {3, 10, 0},
    {5, 11, 0},
    {11, 11, 0},
    {13, 10, 0},
    {14, 8, 0},
    {14, 7, 0},
    {13, 5, 0},
    {11, 4, 0},
```

```
    {8, 4, 0},
    {6, 5, 0},
    {5, 7, 0},
    {5, 8, 0},
    {6, 9, 0},
    {10, 9, 0},
    {11, 8, 0},
    {11, 6, -1},
};

// コンストラクタ
CBallOnRailStage::CBallOnRailStage() : CStage(L"BALL ON RAIL") {

    // 軌道の長さを計算する
    Rail=BallOnRailRail;
    int i;
    for (i=0; Rail[i].Length>=0; i++) {
        float
            x=Rail[i].X-Rail[i+1].X,
            y=Rail[i].Y-Rail[i+1].Y;
        Rail[i].Length=sqrtf(x*x+y*y);
    }

    // 軌道の数
    RailCount=i;

    // 砲台の座標
    SX=8;
    SY=6;
}

// 移動処理
bool CBallOnRailStage::Move(const CInputState* is) {

    // ...（中略）...

    // ステージ開始時はボールを早く進ませて、
    // 一定時間が経過したらゆっくりにする
    Time++;
    if (Time==180) {
        Speed=0.01f;
    }

    // 末尾のボールを少し進ませる
    Ball[0].Pos+=Speed;

    // ボールの間隔を調整する
    for (int i=1; i<BallCount; i++) {

        // あるボールと、
```

```
        // 1つ前の（スタートに近い）ボールについて、
        // 間の距離を調べる
        BALL_ON_RAIL_BALL &b0=Ball[i-1], &b1=Ball[i];
        float d=b1.Pos-b0.Pos;

        // 距離がボール1個未満ならば、
        // ボール1個の距離まで、ボールを少しずつ遠ざける
        if (d<1) {
            b1.Pos=min(b1.Pos+0.2f, b0.Pos+1);
        } else

        // 距離がボール1個より大きければ、
        // ボール1個の距離まで、ボールを少しずつ近づける
        if (d>1) {
            b1.Pos=max(b1.Pos-0.2f, b0.Pos+1);
        }
    }

    // ボールの座標を更新する
    for (int i=0; i<BallCount; i++) {

        // 軌道上の位置からボールの座標を計算する
        SetBallXY(i);

        // ...（中略）...
    }

    // ...（中略）...
}

// ボールの座標を計算する処理
void CBallOnRailStage::SetBallXY(int index) {

    // ボールの位置から、軌道の区画の長さを減算していく
    BALL_ON_RAIL_BALL &b=Ball[index];
    float pos=b.Pos;
    int i;
    for (i=0; i<RailCount && pos>=Rail[i].Length; i++) {
        pos-=Rail[i].Length;
    }

    // ボールの位置よりも区画の長さが大きくなったら、
    // ボールがどの区画にあるのかがわかるので、
    // 区画内の座標を計算する
    if (i<RailCount) {

        // 区画と次の区画の座標を、
        // ボールの位置で比例配分することによって、
        // ボールの座標を計算する
        BALL_ON_RAIL_RAIL &r0=Rail[i], &r1=Rail[i+1];
```

```
            b.X=r0.X+(r1.X-r0.X)*pos/r0.Length;
            b.Y=r0.Y+(r1.Y-r0.Y)*pos/r0.Length;
        } else

        //ボールがどの区画上にもないときには、
        //ボールを削除する
        {
            EraseBall(index);
        }
    }

//描画処理
void CBallOnRailStage::Draw() {

    //画面の解像度に応じて描画サイズを決める
    float
        sw=Game->GetGraphics()->GetWidth()/MAX_X,
        sh=Game->GetGraphics()->GetHeight()/MAX_Y;

    //軌道の描画
    for (int i=0; i<RailCount; i++) {
        BALL_ON_RAIL_RAIL
            &r0=Rail[i],
            &r1=Rail[i+1];
        float
            dx=(r0.X-r1.X)/r0.Length*0.2f,
            dy=(r0.Y-r1.Y)/r0.Length*0.2f;

        //太い線分を描画する
        Game->Texture[TEX_FILL]->Draw(
            (r0.X-dy)*sw, (r0.Y+dx)*sh, COL_BLACK, 0, 0,
            (r0.X+dy)*sw, (r0.Y-dx)*sh, COL_BLACK, 0, 1,
            (r1.X-dy)*sw, (r1.Y+dx)*sh, COL_BLACK, 1, 0,
            (r1.X+dy)*sw, (r1.Y-dx)*sh, COL_BLACK, 1, 1
        );
    }

    //ボールの描画
    for (int i=0; i<BallCount; i++) {
        BALL_ON_RAIL_BALL &b=Ball[i];
        float
            x=(b.X-0.5f)*sw,
            y=(b.Y-0.5f)*sh;

        //ボールの背後を白く塗りつぶす
        Game->Texture[TEX_ORB]->Draw(
            x, y, sw, sh, 0, 0, 1, 1, D3DXCOLOR(1, 1, 1, b.Alpha));

        //ボールの種類に応じたグラフィックを描画する
        Game->Texture[TEX_ORB0+b.Type]->Draw(
```

```
            x, y, sw, sh, 0, 0, 1, 1, D3DXCOLOR(0, 0, 0, b.Alpha));
    }

    // ...(中略)...
}
```

SAMPLE

「BALL ON RAIL」は「軌道に沿って進むボール」「新しいボールを軌道上に追加する」「ボールを任意の方向に撃つ」「ボールを軌道に撃ち込む」「軌道上に並んだボールを消す」のサンプルです。

レバーの左右(カーソルキーの左右)で砲台が回転します。ボタン0(Zキー)を押すと、ボールを撃つことができます。

軌道に沿って多数のボールが進んできます。撃ったボールが軌道上のボールに接触すると、軌道上にボールを追加することができます。

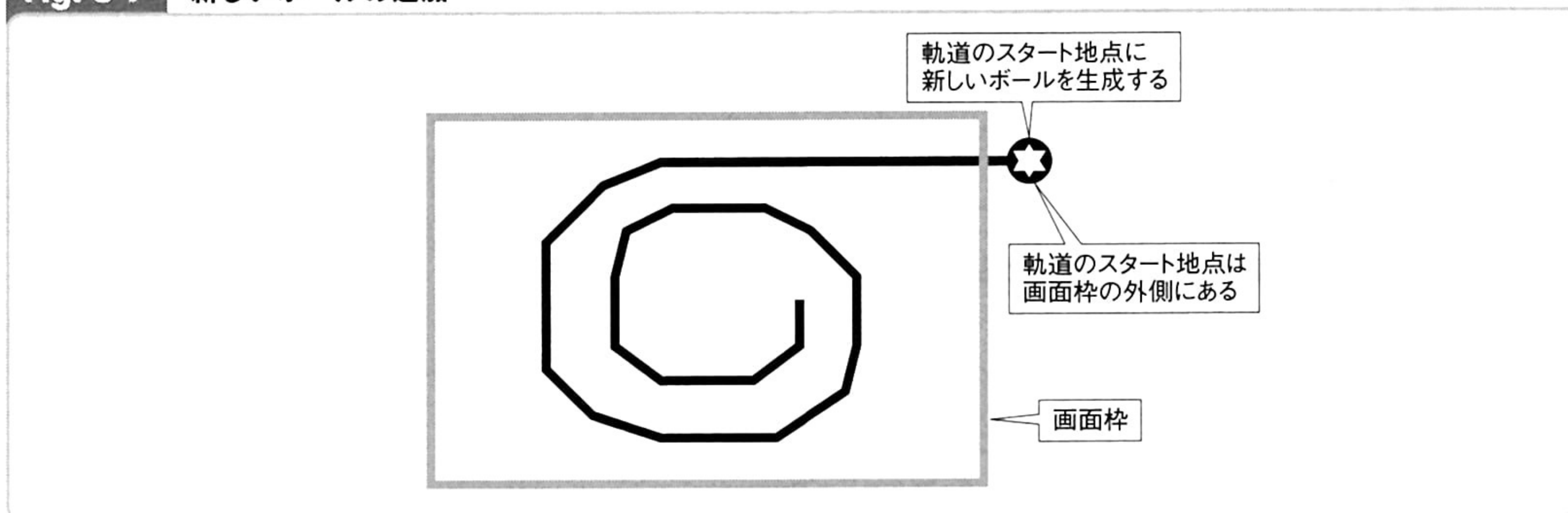
軌道上に同じ種類のボールが規定数(3個)以上並ぶと、ボールを消すことができます。ボールが消えて、ボールの列に空きができると、空きを埋めるように列の先頭部分が後退します。空きが埋まったときに、再び同じ種類のボールが規定数以上並ぶと、連鎖的に消えます。

BALL ON RAIL → p. 389

新しいボールを軌道上に追加する

「軌道に沿って進むボール」(→p. 278)について、軌道のスタート地点に新しいボールを追加するアクションです。新しいボールを追加することによって、軌道上のボールの列が長くなっていきます。

新しいボールは軌道のスタート地点に追加します(Fig. 5-7)。このとき、ボールをスムーズに画面外から進入させるために、スタート地点は画面枠の少し外側にしておくとよいでしょう。

Fig. 5-7 新しいボールの追加

新しいボールの種類は、ランダムに決まります。ただし、同じ種類のボールばかりが連続しないように調整します。

アルゴリズム

新しいボールを軌道上に追加するには、まず先行するボールの位置を調べます (Fig. 5-8)。スタートに一番近いボールが、スタートからボール1個分離れていたら、スタート地点に新しいボールを追加します。ボール1個分離れていなかったら、ボールが重なってしまうので、既存のボールが十分に離れるのを待ちます。

本書では、ボールの列を配列で管理しています (Fig. 5-9)。スタートに最も近いボールを先頭にして、スタートに近い順にボールの情報を格納します。ボールの情報は次の4つです。

- 種類
- 位置
- 座標
- 不透明度

新しいボールを追加するときには、スタートに最も近いので、配列の先頭に挿入します (Fig. 5-10)。配列の要素を1個ずつ後方へ動かし、先頭の要素に新しいボールの情報を書き込みます。

新しいボールの種類はランダムに決めますが、同じ種類ばかりが続いてしまうと不都合です (Fig. 5-11)。それは、「軌道上で並んだボールを消す」(→p. 297) で解説するように、同じ種類のボールが規定数 (3個) 以上並ぶと、出現した瞬間にボールが消えてしまうためです。そこで、ボールの配列の先頭を調べて、先頭に同じ種類のボールが「規定数-1 (2個)」以上並んでいるときには、新しいボールを異なる種類にします。

Fig. 5-8 先行するボールの位置を調べる

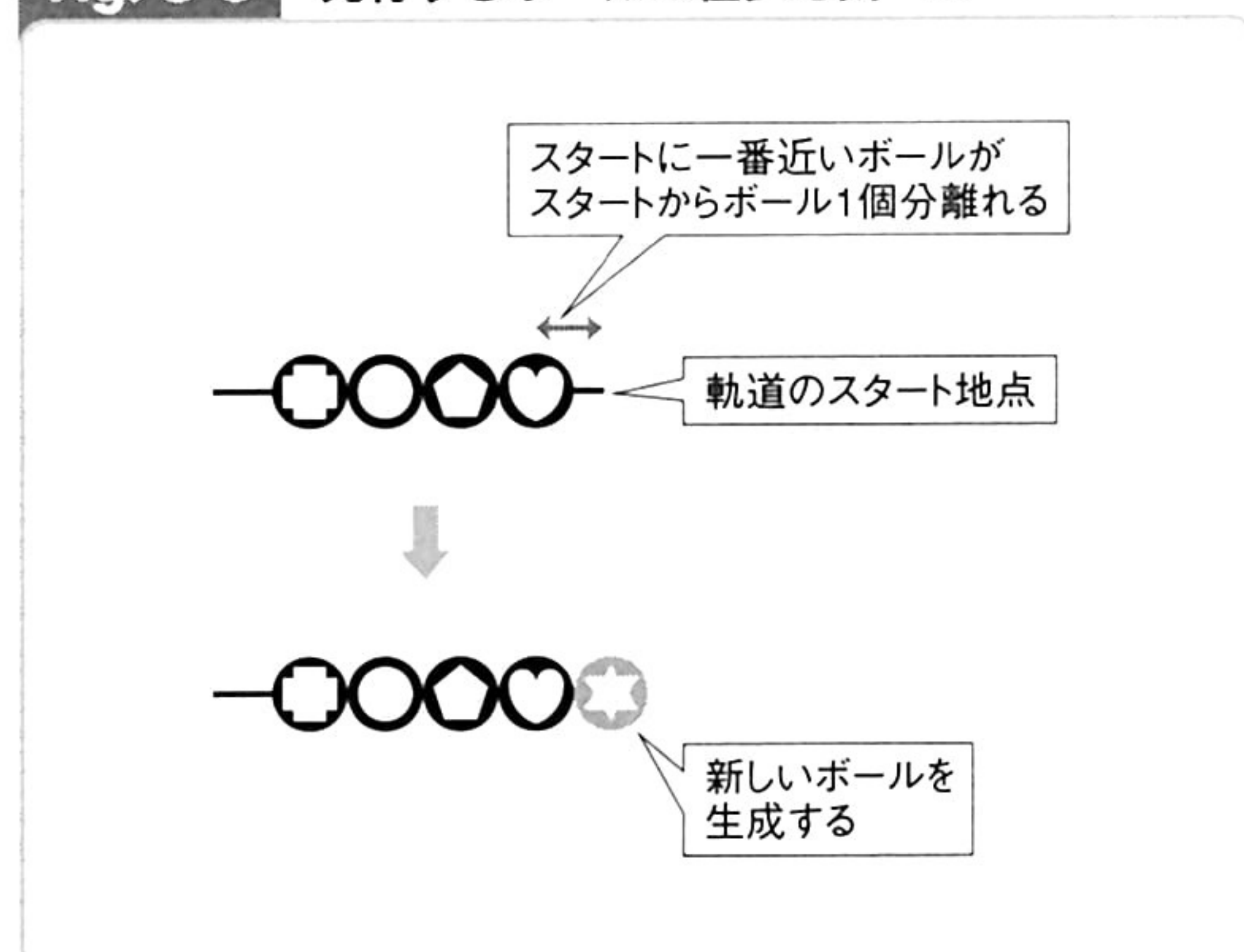



Fig. 5-9 ボールの配列

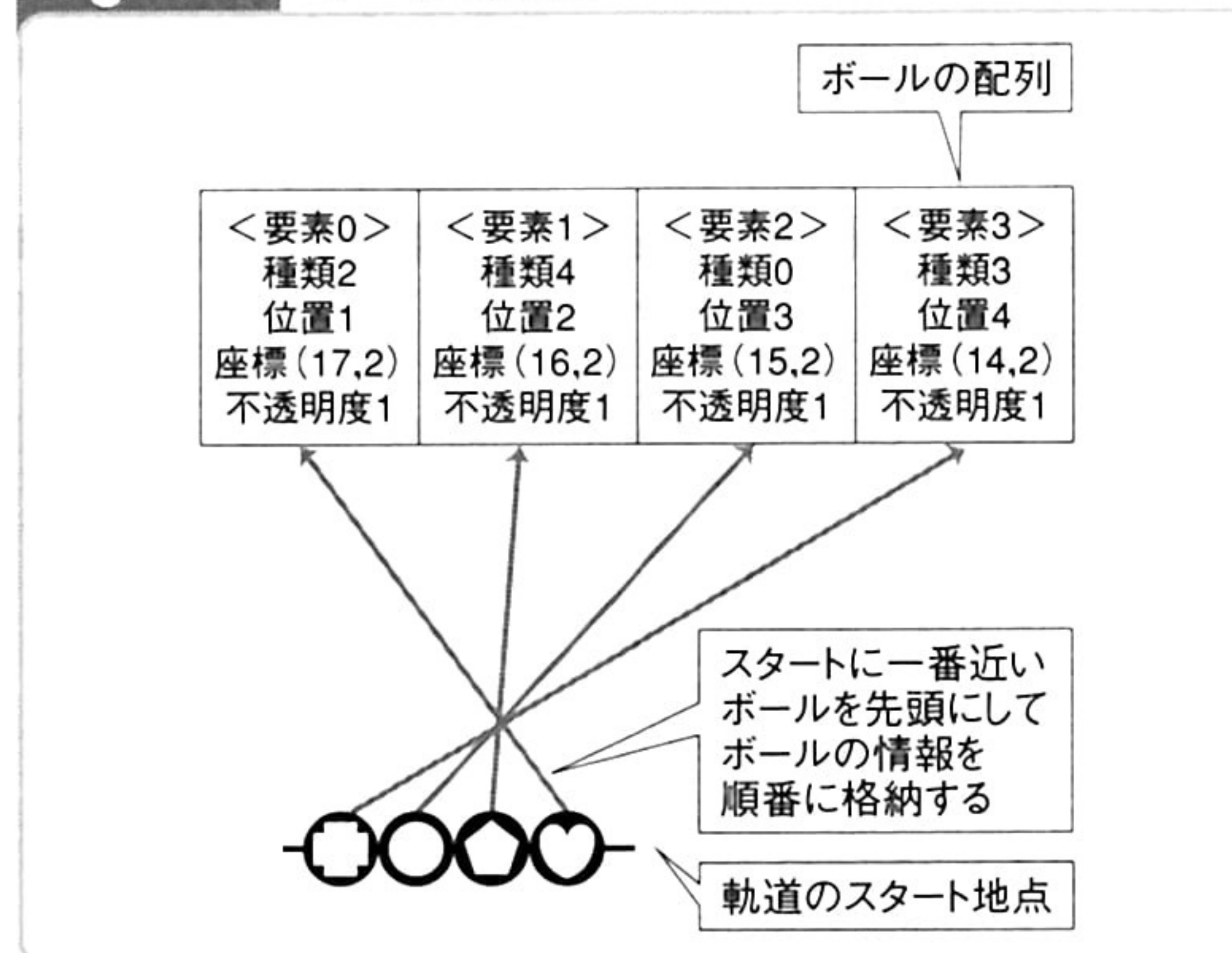



Fig. 5-10 新しいボールを配列に追加する

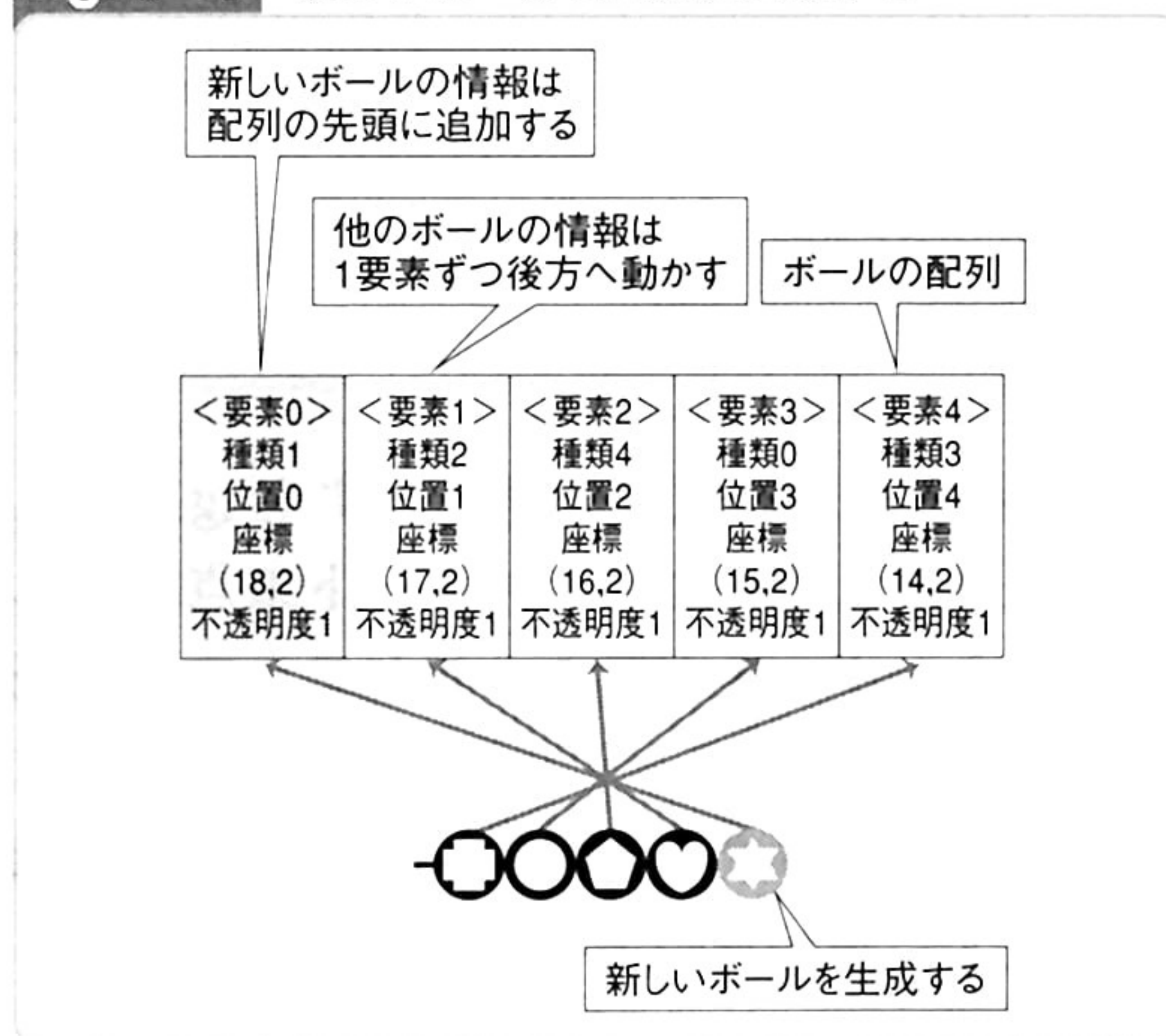
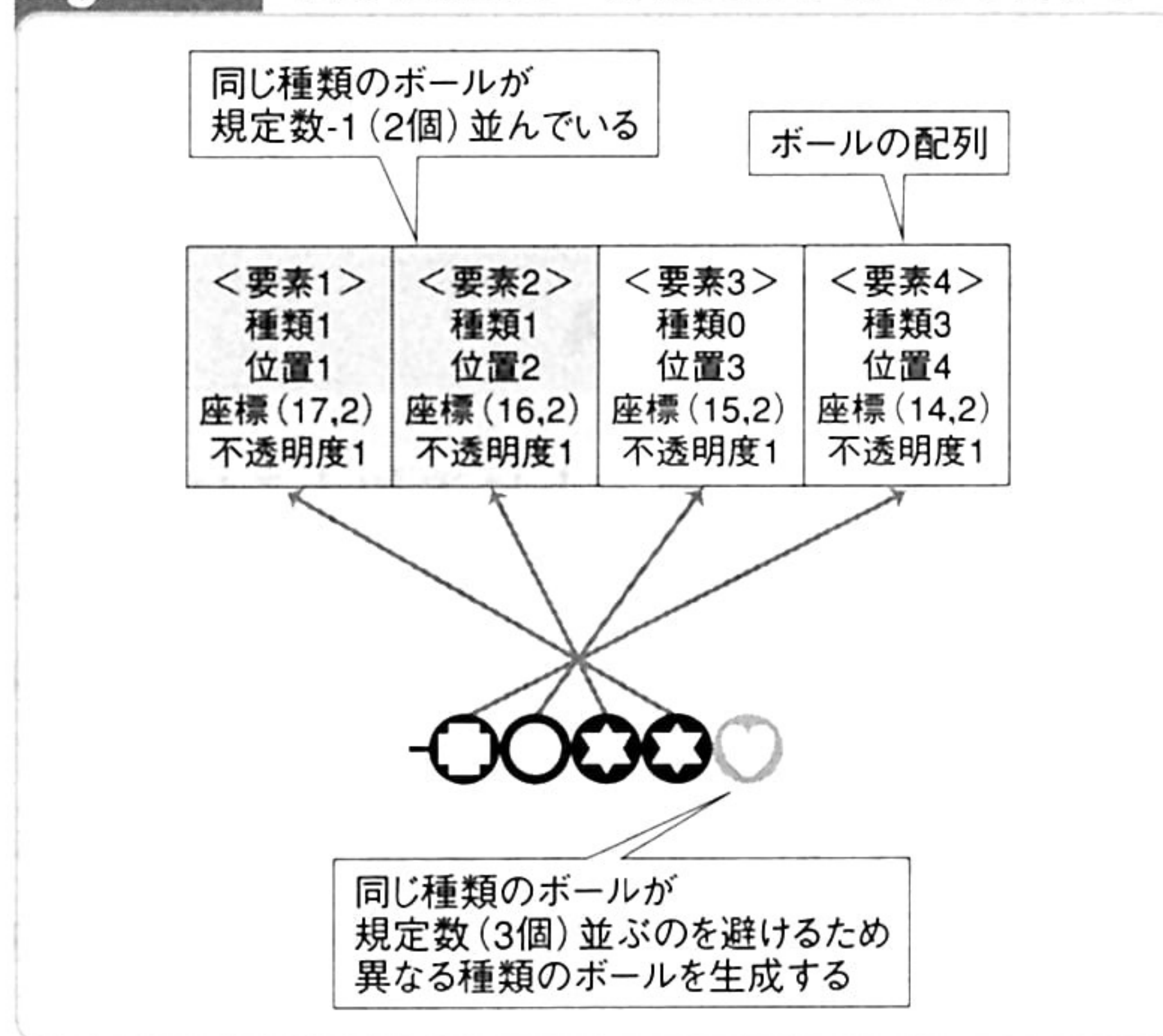



Fig. 5-11 同じ種類のボールが連続しないようにする

プログラム

List 5-2は新しいボールを軌道上に追加するプログラムです。ステージの移動処理と、ボールを追加する処理を掲載しました。

移動処理（Move関数）では、スタートに一番近いボールの位置を調べます。位置がスタートからボール1個分以上離れているか、軌道上にボールがまったくなければ、新しいボールを追加します。

ボールの種類はランダムに決めます。ただし、スタートに同じ種類のボールが規定数-1（2個）以上連続しているときには、ボールが規定数（3個）以上並ぶのを避けるために、新しいボールは異なる種類にします。同じ種類になったときには、ランダムに種類を決め直します。

種類が決まったら、ボールを追加する処理（AddBall関数）を呼び出します。ここではまず、ボールの配列（Ball）の指定された位置以降の要素を、1要素ずつ後方に動かします。そして、指定された位置に新しい要素を書き込みます。

List 5-2 新しいボールを軌道上に追加する（CBallOnRailStageクラス）

```
// 移動処理
bool CBallOnRailStage::Move(const CInputState* is) {

    // ステージ内のボールの数が0個か、
    // スタートから一番近いボールまでの距離がボール1個分以上ならば、
    // 新しいボールをスタートに生成する
    if (
        (BallCount==0 || Ball[0].Pos>=1) &&
        BallCount<BALL_ON_RAIL_COUNT
    ) {
```

```
        // ボールの種類をランダムに決める
        int type=Rand.Int31()%BALL_ON_RAIL_TYPE;

        // 同じ種類のボールを規定数（3個）以上連続させないための処理
        if (BallCount>=BALL_ON_RAIL_ERASE-1) {

            // スタートに同じ種類のボールが規定数-1（3-1=2個）
            // 連続しているかどうかを調べる
            int i;
            for (i=1; i<BALL_ON_RAIL_ERASE-1; i++) {
                if (Ball[i].Type!=Ball[0].Type) break;
            }

            // 連続している場合、
            // 異なる種類のボールを生成する
            if (i==BALL_ON_RAIL_ERASE-1) {

                // 異なる種類のボールをランダムに選ぶ
                if (type==Ball[0].Type) {
                    type+=Rand.Int31()%(BALL_ON_RAIL_TYPE-1)+1;
                    type%=BALL_ON_RAIL_TYPE;
                }
            }
        }

        // ボールを追加する
        AddBall(0, type, 0);
    }

    // …（中略）…
}

// ボールを追加する処理
void CBallOnRailStage::AddBall(int index, int type, float pos) {

    // ボールが最大数に達していなければ、ボールを追加する
    if (BallCount<BALL_ON_RAIL_COUNT) {

        // 配列上で追加する位置よりも後のボールを、
        // 1個ずつ後方にずらす
        for (int i=BallCount; i>index; i--) {
            Ball[i]=Ball[i-1];
        }

        // ボールの数を増やす
        BallCount++;

        // ボールの種類・位置・不透明度を設定する
        BALL_ON_RAIL_BALL &b=Ball[index];
        b.Type=type;
```

```
            b.Pos=pos;
            b.Alpha=1;

            // ボールの座標を計算する
            SetBallXY(index);
        }
    }
```

ボールを任意の方向に撃つ

「軌道に沿って進むボール」(→p. 278) に関連して、砲台からボールを撃つアクションです。砲台はレバー操作で左右に回転させることができます。

砲台はステージ上に配置されています (Fig. 5-12)。本書のサンプルではステージの中央付近に置かれていますが、中央以外に配置してもよいでしょう。

レバーを左右に入力すると、砲台は左右に回転します。砲台の中央に表示されているのは、これから撃つボールです。

ボタンを押すと、ボールを撃つことができます (Fig. 5-13)。ボールは砲台が向いている方向に飛んでいきます。

撃ったボールが軌道上のボールに当たると、軌道上にボールを撃ち込むことができます。詳細は「ボールを軌道に撃ち込む」(→p. 294) で解説します。軌道上のボールに当たらずに、画面外に出ると、撃ったボールは消滅します。撃ち込んだ場合も、消滅した場合も、砲台には新しいボールが出現します。

ボールを任意の方向に撃つアクションは『パズループ』に採用されています。特にアーケード版では、ダイヤルコントローラで砲台を回転させることができ、独特の操作感が楽しめます。

Fig. 5-12 砲台

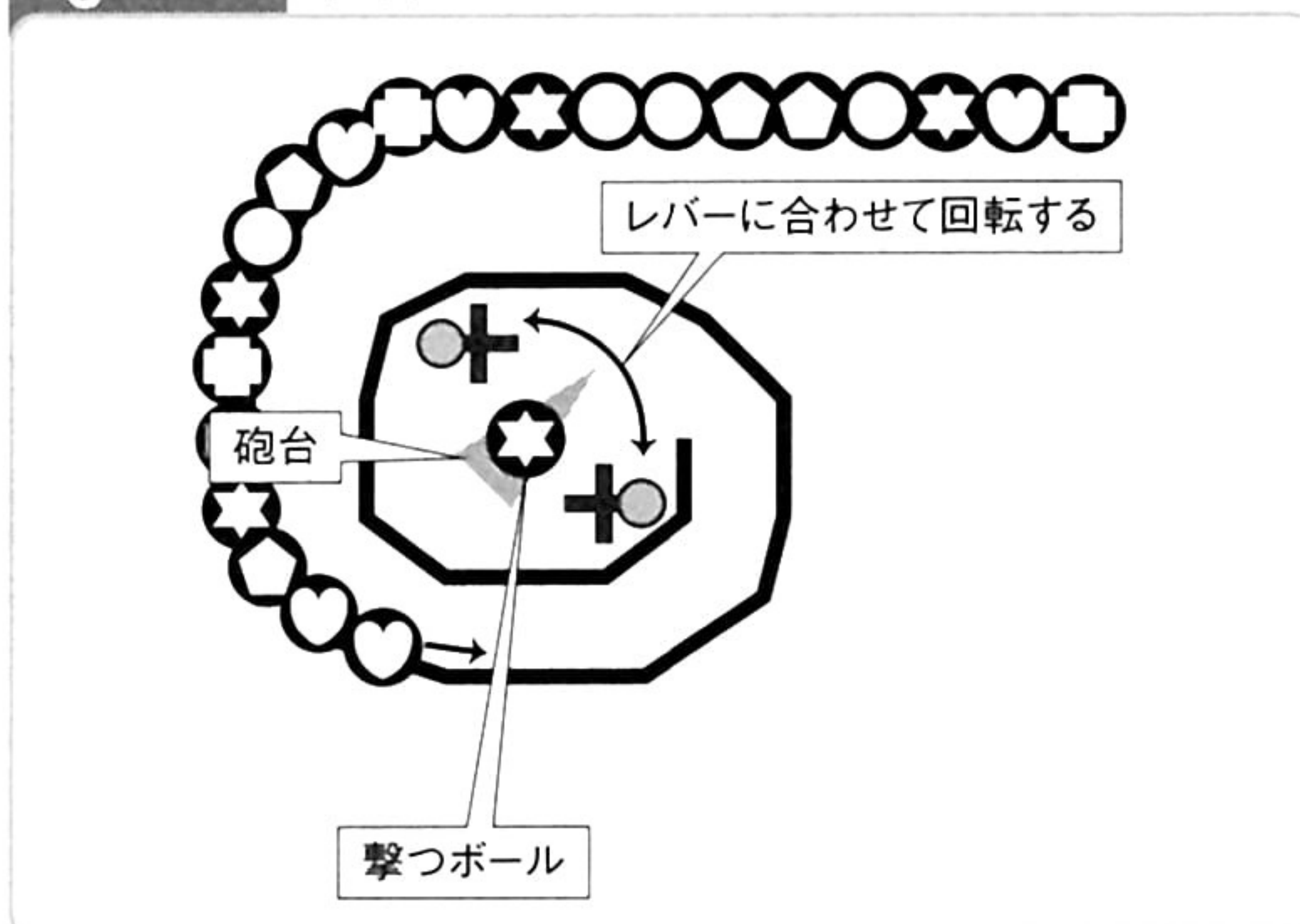
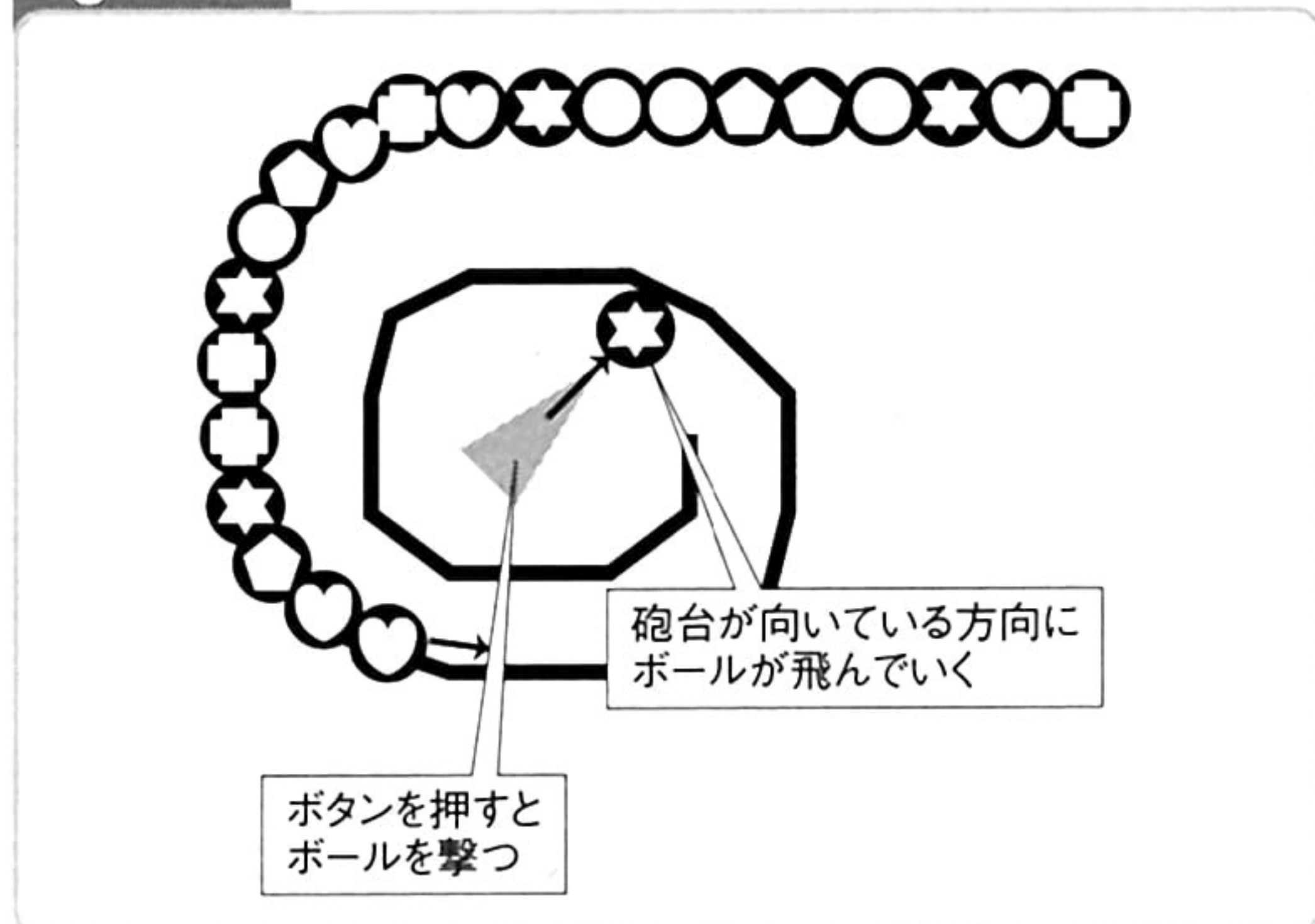



Fig. 5-13 ボールを撃つ

アルゴリズム

ボールを任意の方向に撃つには、まず砲台の回転について考えます。レバー入力に応じて、砲台の角度を変化させます（Fig. 5-14）。レバーを左に入れたら左回りに、右に入れたら右回りに、砲台を回転させます。

本書のサンプルでは、X軸の方向を「0.0」として、「1.0」が360度に相当するような角度を用いています。角度を減少させると左回りに、増加させると右回りになります。レバーの向きに合わせて角度の値を増減させれば、砲台を回転させることができます。

ボタンを押したら、砲台からボールを発射します。砲台が向いている方向にボールが飛ぶように、ボールの速度を設定します（Fig. 5-15）。砲台の角度をSDir（0.0～1.0）、ボールの速さをBSpeed、円周率をPIとすると、ボールの速度（BVX, BVY）は、

BVX=cos（SDir×PI×2）×BSpeed
BVY=sin（SDir×PI×2）×BSpeed

となります。

Fig. 5-14 砲台の角度を変化させる

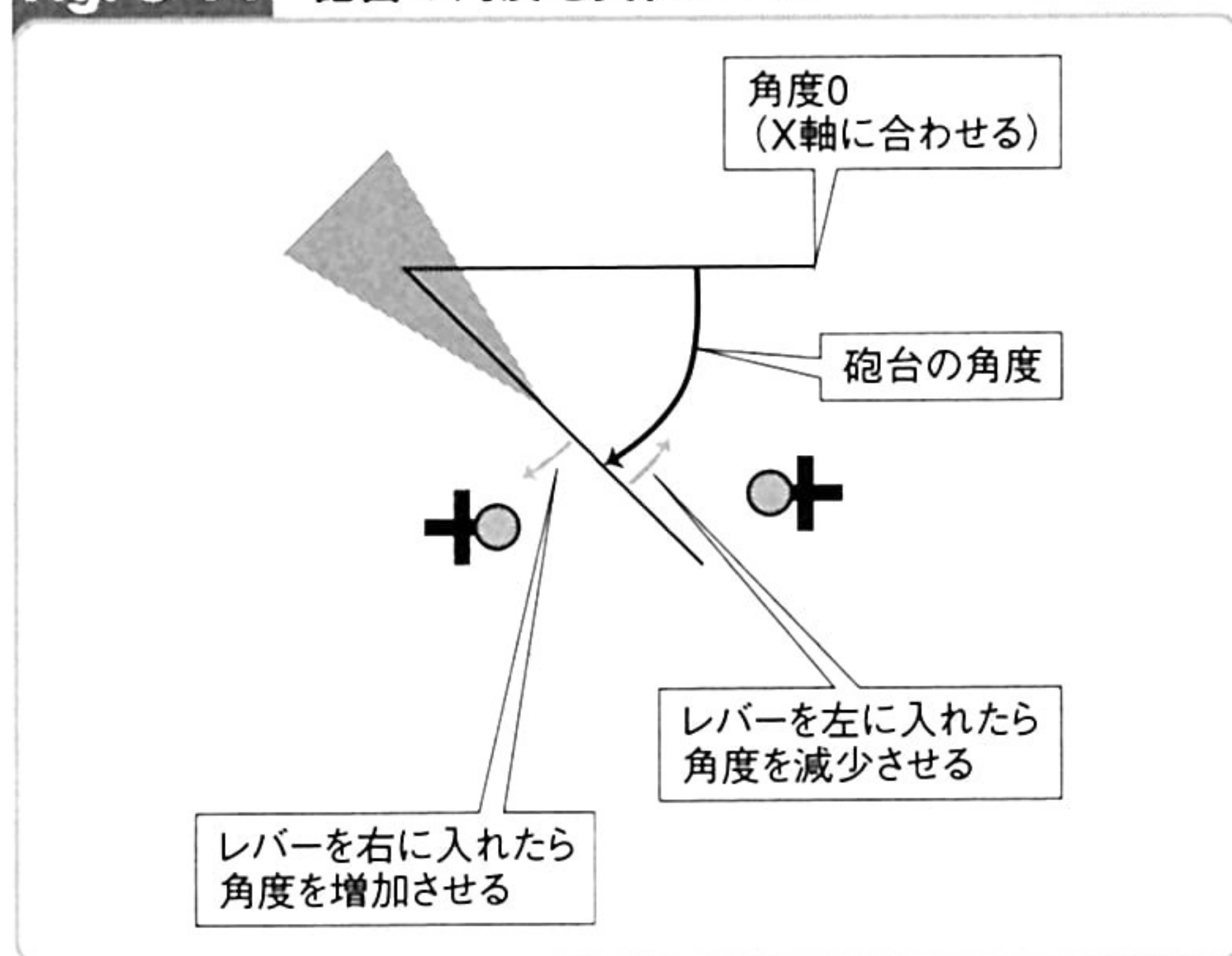
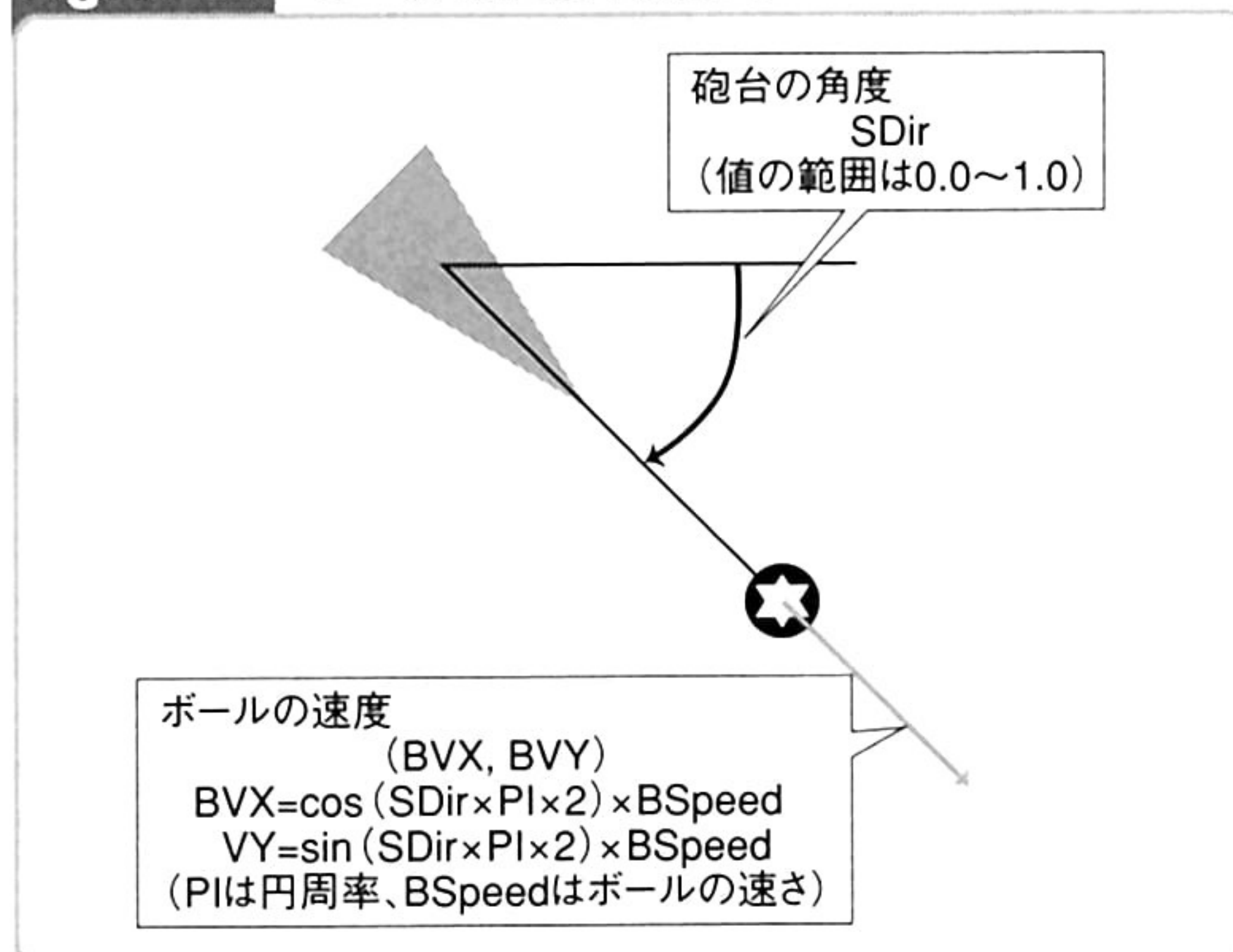



Fig. 5-15 ボールの速度を決める

プログラム

List 5-3はボールを任意の方向に撃つプログラムです。ステージの移動処理、撃つボールを作成する処理、ステージの描画処理を掲載しました。

移動処理（Move関数）では、レバー入力に応じて砲台を左右に回転させます。ボールが砲台上にあるときにボタンを押したら、ボールを撃ちます。砲台の角度に応じて、ボールの速度を設定します。

ボールが飛んでいる間は、ボールの座標を更新します。もしもボールが画面外に出たら、ボールを削除して、新しく撃つボールを作ります。撃つボールを生成する処理（InitBall関数）では、砲台と同じ位置に、ランダムな種類のボールを生成します。

撃ったボールが軌道上のボールに接触したら、ボールを軌道上に撃ち込む処理を行います。詳細は「ボールを軌道に撃ち込む」（→p. 294）で解説します。

描画処理（Draw関数）では、砲台とボールを描画します。砲台に関しては、三角形を砲台の角度に応じて回転して描画することで表現しました。砲台のグラフィックを、角度に応じて回転して描画してもよいでしょう。撃つボールに関しては、軌道上のボールと同様に、種類に応じたグラフィックを描画します。

List 5-3　ボールを任意の方向に撃つ（CBallOnRailStageクラス）

```
// 移動処理
bool CBallOnRailStage::Move(const CInputState* is) {

    // ...（中略）...

    // レバー入力に応じて、砲台を左右に回転させる
    if (is->Left) SDir+=0.99f;
    if (is->Right) SDir+=0.01f;
    if (SDir>1) SDir-=(int)SDir;

    // ボールを撃てる状態でボタンを押したら、ボールを撃つ
    if (BallReady) {
        if (!PrevButton && is->Button[0]) {

            // 砲台の方向に応じて、ボールの速度を決める
            float rad=SDir*D3DX_PI*2;
            BVX=cosf(rad)*0.5f;
            BVY=sinf(rad)*0.5f;

            // ボールを撃てない状態にする
            BallReady=false;
        }
    } else

    // ボールが飛んでいる間の処理
    {
        // ボールの座標を更新する
        BX+=BVX;
        BY+=BVY;

        // ボールが画面外に出たら消去して、新しいボールを生成する
        if (
            BX<-1 || BX>=MAX_X+1 ||
            BY<-1 || BY>=MAX_Y+1
        ) {
```

```
            InitBall();
        } else

        // ボールが軌道上のボールにぶつかったかどうかを調べる
        {
            TryAddBall();
        }
    }

    // ボタンを押しっぱなしにしたときに、
    // 連続してボールを撃たないため、
    // ボタンの入力状態を記録する
    PrevButton=is->Button[0];

    // ...(中略)...
}

// 撃つボールを生成する処理
void CBallOnRailStage::InitBall() {

    // 座標は砲台の座標に合わせる
    BX=SX;
    BY=SY;

    // 種類はランダムに決める
    BType=Rand.Int31()%BALL_ON_RAIL_TYPE;

    // ボールを撃てる状態にする
    BallReady=true;
}

// 描画処理
void CBallOnRailStage::Draw() {

    // ...(中略)...

    // 砲台の描画
    float
        rad=SDir*D3DX_PI*2,
        c=cosf(rad), s=sinf(rad),
        f0=2.0f, f1=1.0f, f2=0.5f;
    Game->Texture[TEX_FILL]->Draw(
        (SX+c*f0)*sw, (SY+s*f0)*sh, COL_LGRAY, 0, 0,
        (SX+c*f0)*sw, (SY+s*f0)*sh, COL_LGRAY, 0, 1,
        (SX-c*f1+s*f2)*sw, (SY-s*f1-c*f2)*sh, COL_LGRAY, 1, 0,
        (SX-c*f1-s*f2)*sw, (SY-s*f1+c*f2)*sh, COL_LGRAY, 1, 1
    );

    // 撃ったボールの描画
    Game->Texture[TEX_ORB0+BType]->Draw(
```

```
        (BX-0.5f)*sw, (BY-0.5f)*sh, sw, sh, 0, 0, 1, 1, COL_BLACK);
}
```

ボールを軌道に撃ち込む

「軌道に沿って進むボール」(→p. 278) において、砲台からボールを撃って、軌道上に撃ち込むアクションです。「ボールを任意の方向に撃つ」(→p. 290) で解説したように、砲台の角度を調整することによって、特定の位置を狙ってボールを撃ち込むことができます。

撃ったボールが軌道上のボールに接触すると、軌道上に取り込まれます (Fig. 5-16)。接触したボールの前後どちら側に取り込まれるかは、当たった場所に応じて変わります (Fig. 5-17)。当たり方の微妙な違いで、ボールが思ったとおりの場所に入ったり入らなかったりするのが、このアクションの面白いところです。

Fig. 5-16 撃ったボールが軌道上に取り込まれる

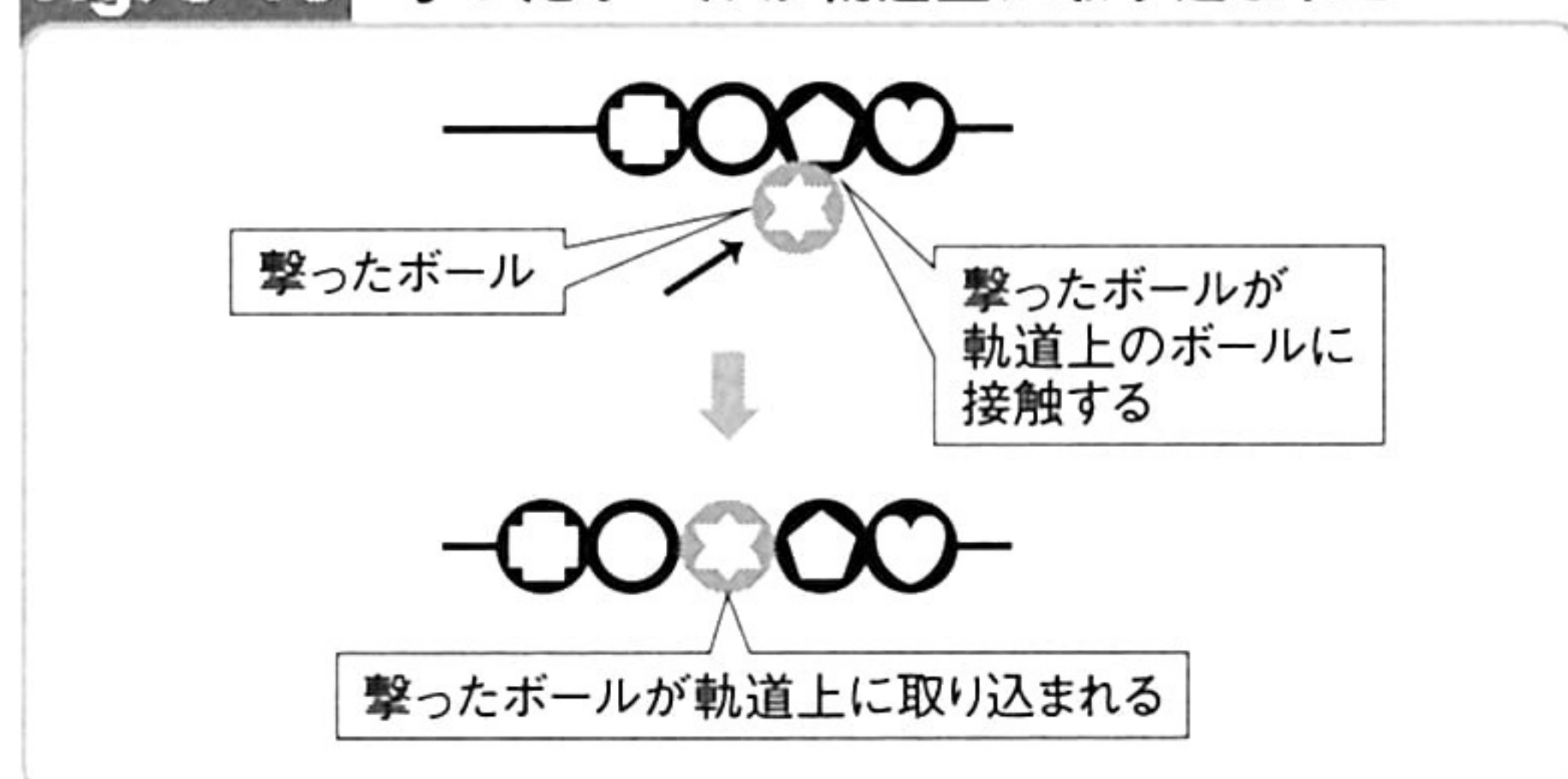
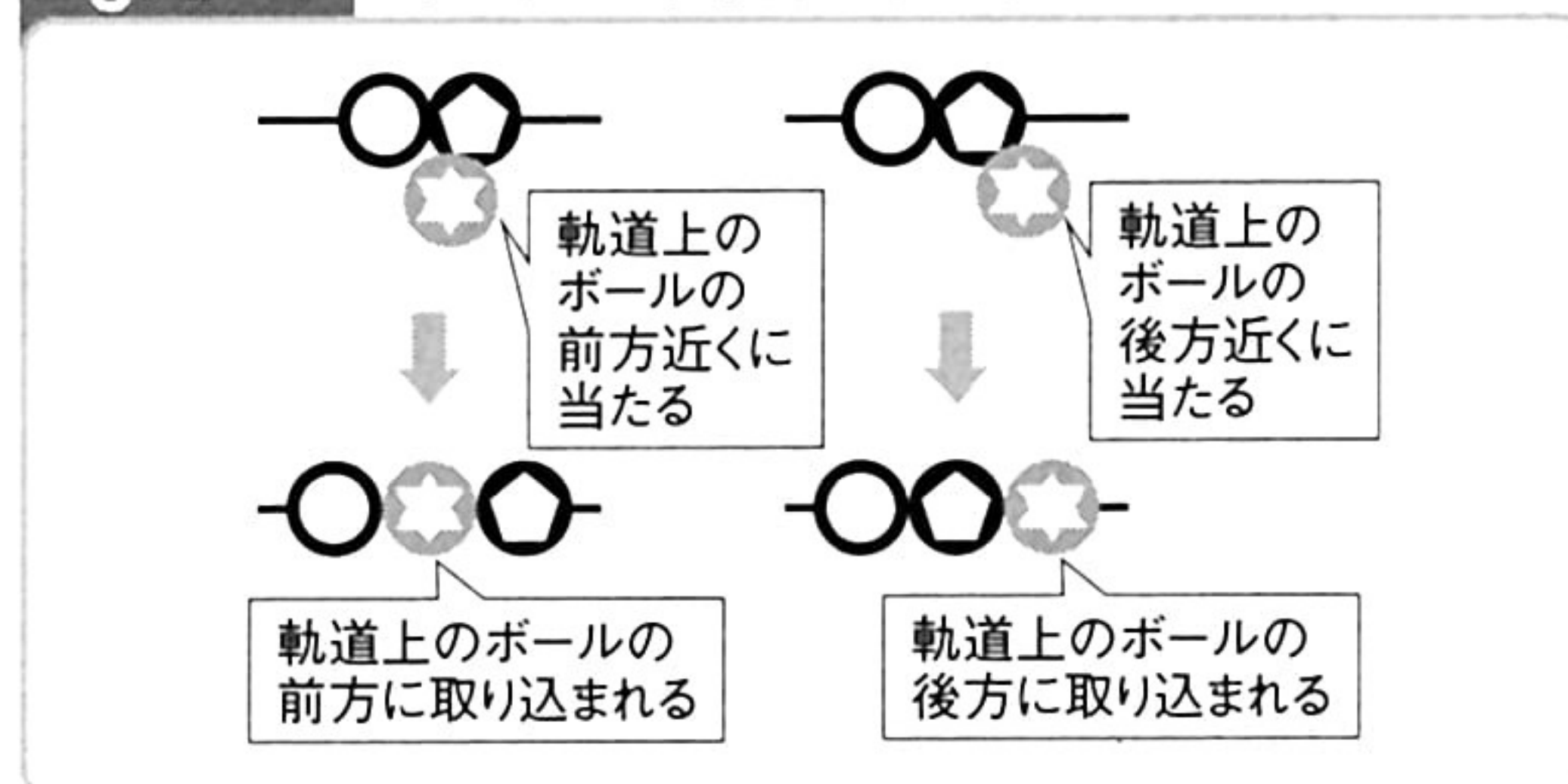
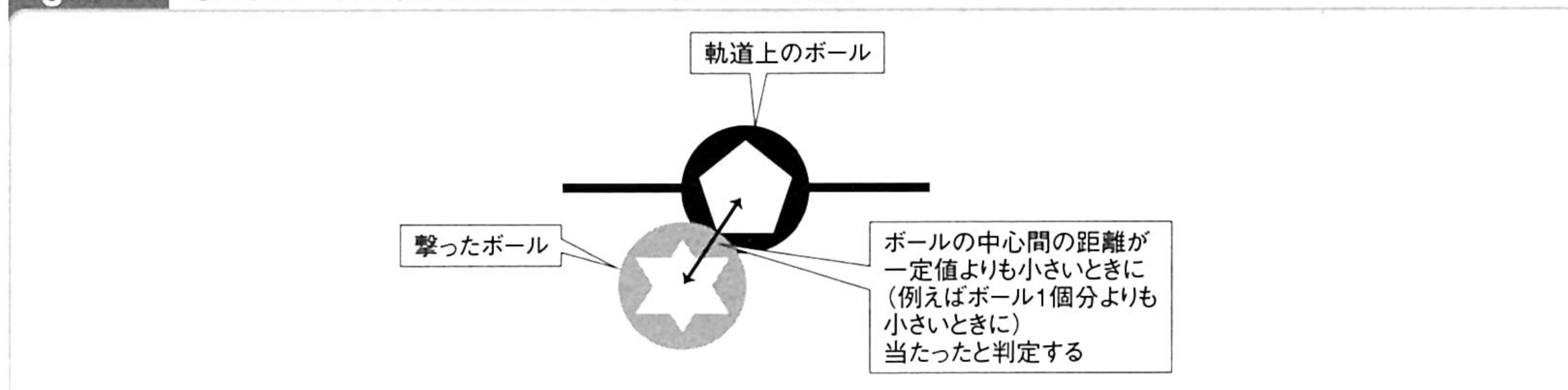



Fig. 5-17 ボールが取り込まれる位置の違い

アルゴリズム

ボールを軌道に撃ち込むには、撃ったボールと軌道上のすべてのボールの間で、当たり判定処理を行います (Fig. 5-18)。丸いボール同士の当たり判定処理なので、中心間の距離を計算し

Fig. 5-18 撃ったボールと軌道上のボールとの当たり判定処理

PuzzleGame Algorithm Maniax

て、距離が一定値よりも小さいときに当たったと判定すればよいでしょう。

本書では距離がボール1個分よりも小さいときに、接触したと判定することにしました。距離を大きめにすると接触しやすく、小さめにすると接触しにくくなります。実際にゲームを遊びながら、快適に遊べるように距離を調整するとよいでしょう。

追加位置を決める

ボール同士が接触したら、撃ったボールを軌道上のボールの前後どちらに追加するのかを決めます (Fig. 5-19)。それには、まずは軌道上のボールが軌道のどの区画上にあるのかを調べます。次に、軌道上のボールの中心を通り、区画の線分に対して垂直な直線を求めます。そして、撃ったボールの中心が軌道にその直線の前後どちら側にあるのかによって、ボールを前後どちらに追加するのかを決めます。

直線の前後どちら側に中心があるのかは、ベクトルの内積を使って判定することができます (Fig. 5-20)。区画の線分の始点から終点に向かうベクトルと、軌道上のボールから撃ったボールの中心に向かうベクトルについて、内積を求めます。

Fig. 5-19 ボールを前後どちらに追加するのかを決める

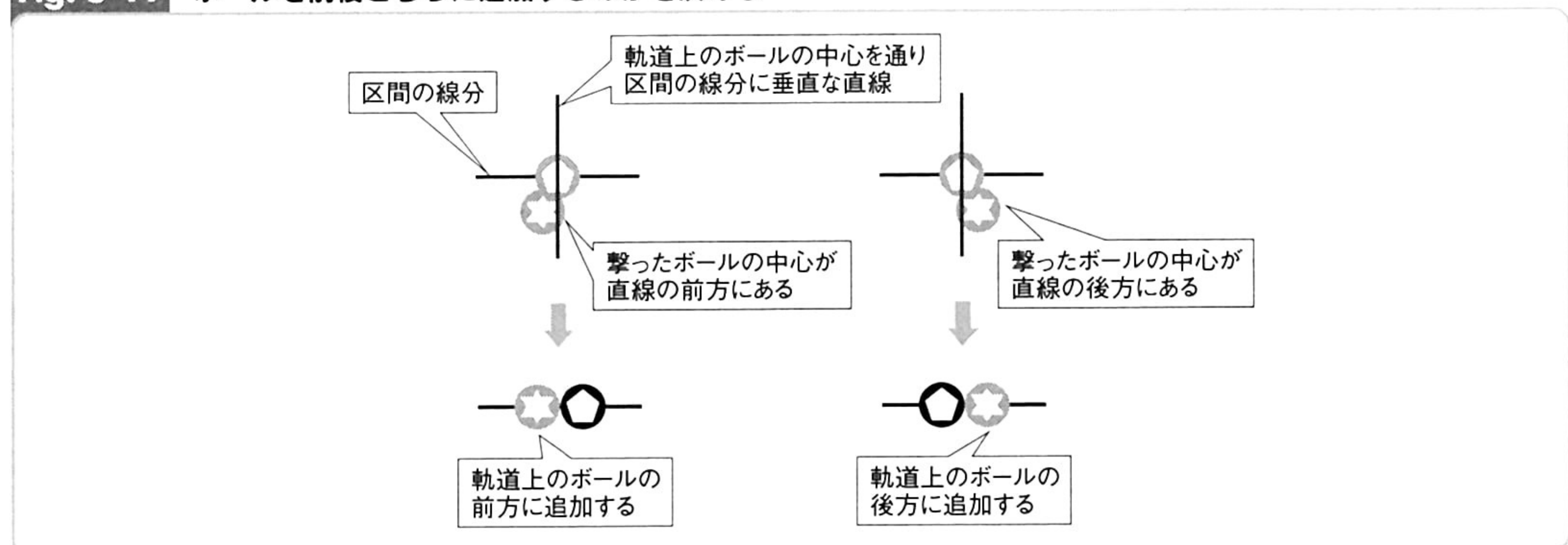
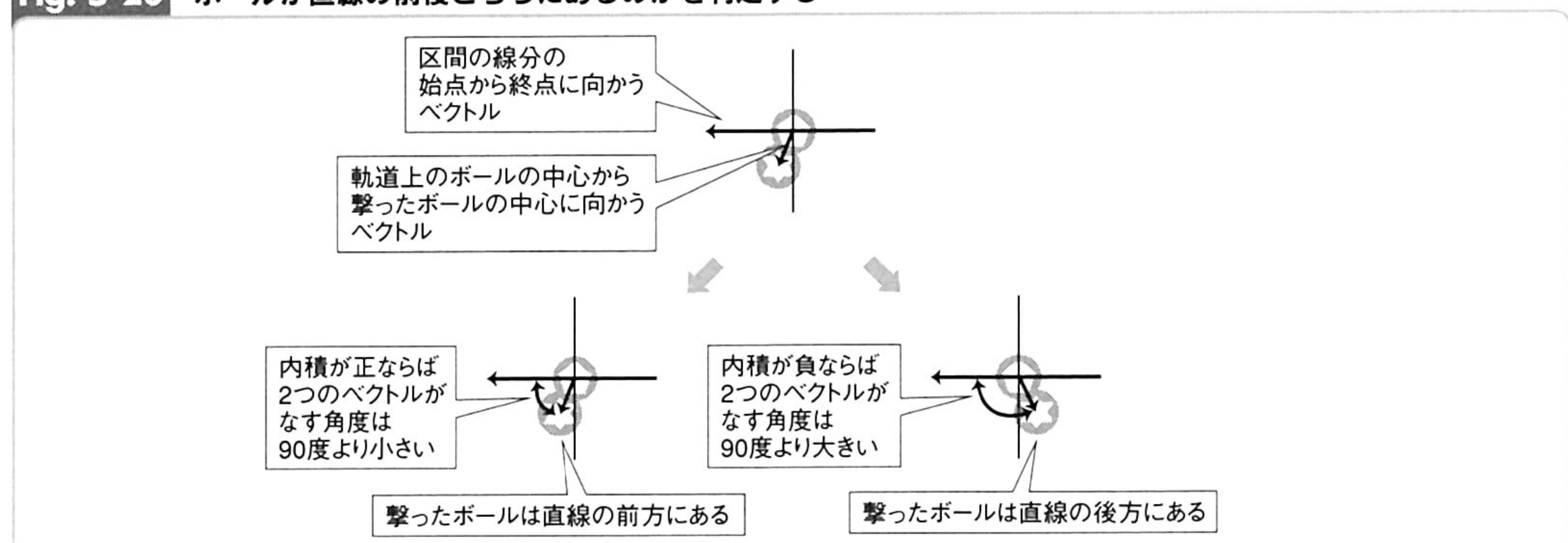



Fig. 5-20 ボールが直線の前後どちらにあるのかを判定する

内積が正の場合には、2つのベクトルのなす角度が90度より小さいということです。この場合、撃ったボールは軌道上のボールの前方に近い側にあるので、前方に追加します。

逆に内積が負の場合には、2つのベクトルのなす角度が90度より大きいということです。この場合、撃ったボールは軌道上のボールの後方に近い側にあるので、後方に追加します。

内積が0の場合には、撃ったボールは軌道上のボールの真横にあるということです。この場合は、前方と後方のどちらに追加しても不自然にはなりません。

プログラム

List 5-4はボールを軌道に撃ち込むプログラムです。ボールを軌道に撃ち込む処理を掲載しました。この処理は、「ボールを任意の方向に撃つ」(→p. 290) のプログラムにおいて、撃ったボールを移動させるたびに呼び出します。

軌道上のすべてのボールについて、撃ったボールとの間で当たり判定処理を行います。ボールの中心間の距離が一定値 (ボール1個分) 未満ならば、ボール同士が接触したと判定します。

接触したら、軌道上のボールがある区画を探します。そして、区画の線分のベクトルと、ボールの中心間を結んだベクトルとの間で、内積を計算します。内積が正か負かによって、撃ったボールを軌道上のボールの前後どちらかに追加します。

List 5-4 ボールを軌道に撃ち込む (CBallOnRailStageクラス)

```
// ボールを軌道に打ち込む処理
void CBallOnRailStage::TryAddBall() {

    // 軌道上のすべてのボールについて調べる
    for (int i=0; i<BallCount; i++) {

        // 撃ったボールと軌道上のボールの距離がボール1個分未満ならば、
        // ボールが接触したと判断し、
        // ボールを軌道上に追加する
        BALL_ON_RAIL_BALL &b=Ball[i];
        float dx=BX-b.X, dy=BY-b.Y;
        if (dx*dx+dy*dy<1) {

            // 軌道上のボールがある区画を探す
            float pos=b.Pos;
            int j;
            for (j=0; j<RailCount && pos>=Rail[j].Length; j++) {
                pos-=Rail[j].Length;
            }
            if (j<RailCount) {

                // 軌道の区画のベクトルと、
                // 撃ったボールと軌道上のボールを結んだベクトルについて、
```

```
                //内積を計算することによって、
                //撃ったボールを軌道上のボールの前後どちらに
                //追加するのかを決める
                BALL_ON_RAIL_RAIL &r0=Rail[j], &r1=Rail[j+1];
                float rx=r1.X-r0.X, ry=r1.Y-r0.Y;

                //軌道上のボールの前方に追加する
                if (dx*rx+dy*ry>0) {
                    AddBall(i+1, BType, b.Pos+0.5f);
                } else

                //軌道上のボールの後方に追加する
                {
                    AddBall(i, BType, b.Pos-0.5f);
                }

                //撃つボールを生成する
                InitBall();
            }
        }
    }
}
```

軌道上に並んだボールを消す

「軌道に沿って進むボール」(→p. 278) において、同じ種類のボールが規定数以上並んだときに、ボールを消すアクションです。「ボールを軌道に撃ち込む」(→p. 294) で解説したように、ボールを軌道に撃ち込んで、同じ種類のボールを揃え、次々に消していくことがゲームの目的です。

軌道上にボールを撃ち込んだときに、同じ種類のボールが規定数 (3個) 以上並ぶと、ボールを消すことができます (Fig. 5-21)。本書のサンプルでは、ボールの表示がだんだん薄くなり、一定時間後に完全に消えます。

ボールが消えると、空いた場所を埋めるように、前方のボールが後退します (Fig. 5-22)。空いた場所が埋まったときに、再び同じ種類のボールが並ぶと、連鎖的にボールを消すことができます。

Fig. 5-21 同じ種類のボールを並べて消す

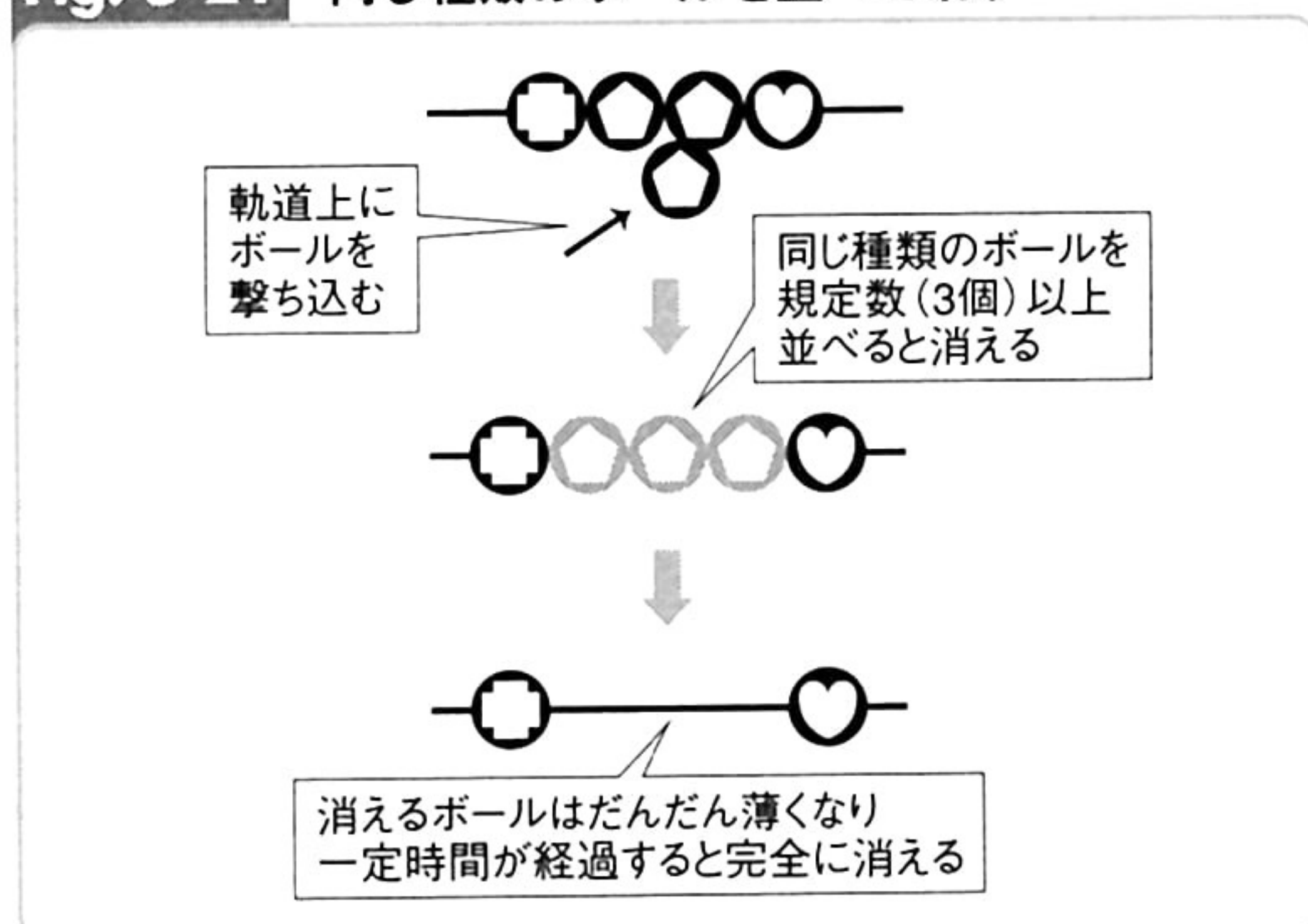
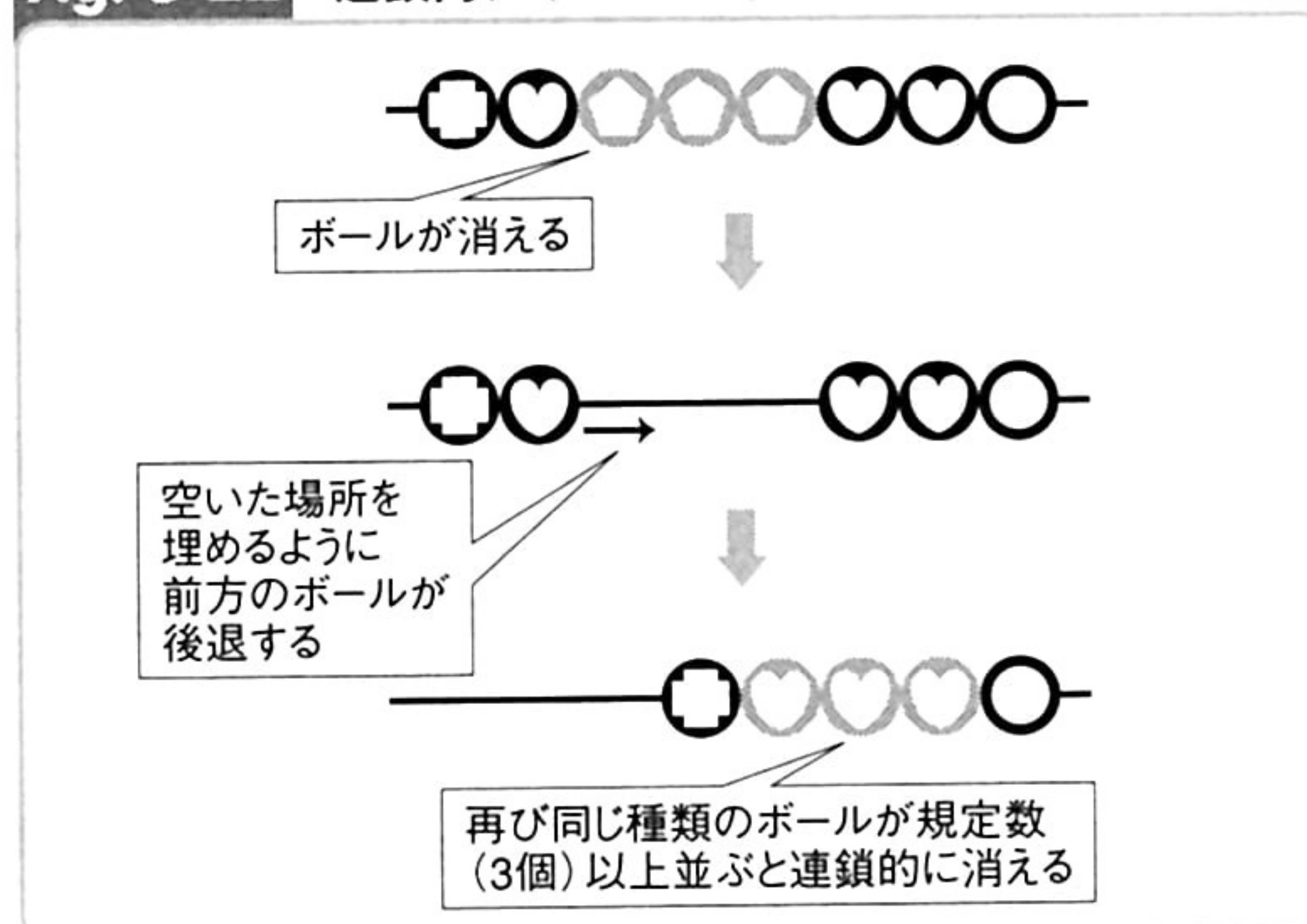
軌道上にボールを撃ち込む

同じ種類のボールを規定数（3個）以上並べると消える

消えるボールはだんだん薄くなり一定時間が経過すると完全に消える

Fig. 5-22 連鎖的にボールを消す

ボールが消える

空いた場所を埋めるように前方のボールが後退する

再び同じ種類のボールが規定数（3個）以上並ぶと連鎖的に消える

アルゴリズム

軌道上に並んだボールを消すには、ボールの配列を調べて、同じ種類のボールが並んでいるかどうかを調べます（Fig. 5-23）。同じ種類のボールが規定数（3個）以上並んでいたら、それらのボールを消します。

ボールの配列を調べるときには、ボールの種類だけではなく、ボール間の距離も調べます。離れているボールは、並んでいるとはみなしません。ボール間の距離がボール1個分のとき、つまりボール同士が接しているときだけ、並んでいるとみなします。

Fig. 5-23 消えるボールを探す

ボールの配列

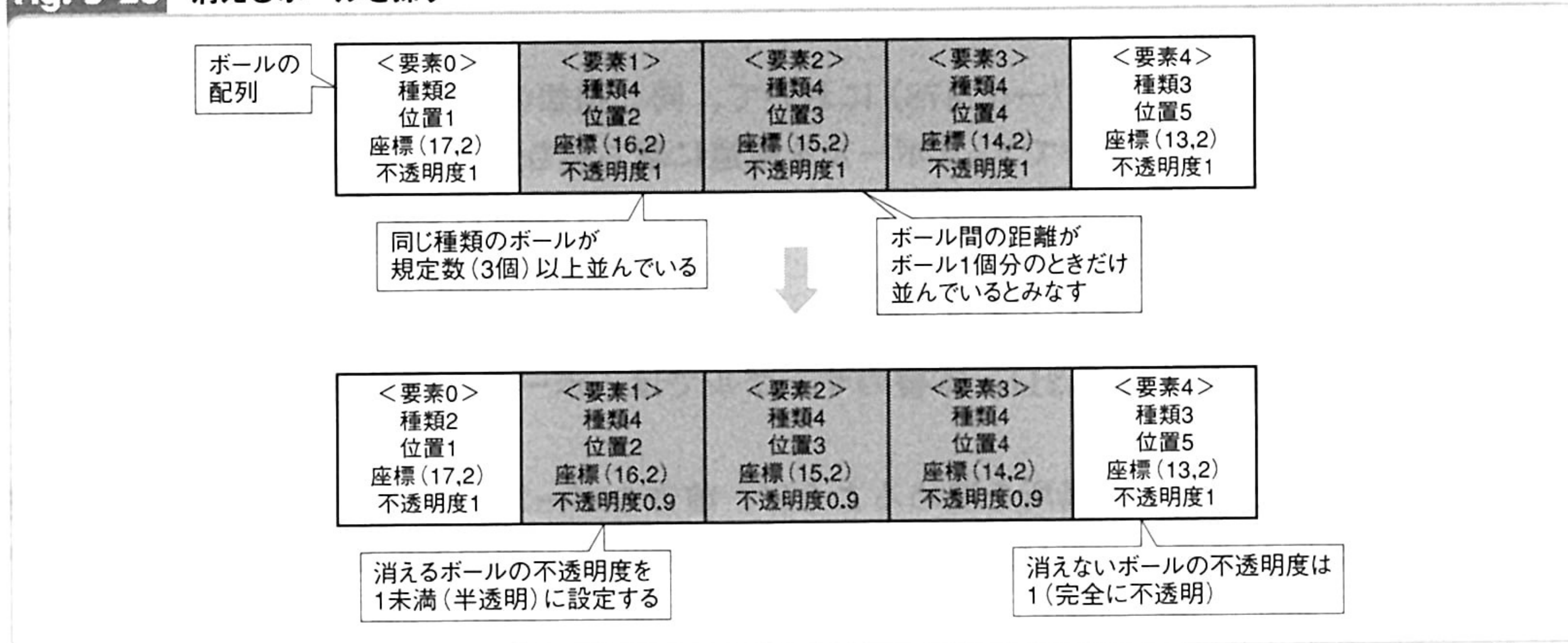
＜要素0＞	＜要素1＞	＜要素2＞	＜要素3＞	＜要素4＞
種類2	種類4	種類4	種類4	種類3
位置1	位置2	位置3	位置4	位置5
座標（17,2）	座標（16,2）	座標（15,2）	座標（14,2）	座標（13,2）
不透明度1	不透明度1	不透明度1	不透明度1	不透明度1

同じ種類のボールが規定数（3個）以上並んでいる

ボール間の距離がボール1個分のときだけ並んでいるとみなす

＜要素0＞	＜要素1＞	＜要素2＞	＜要素3＞	＜要素4＞
種類2	種類4	種類4	種類4	種類3
位置1	位置2	位置3	位置4	位置5
座標（17,2）	座標（16,2）	座標（15,2）	座標（14,2）	座標（13,2）
不透明度1	不透明度0.9	不透明度0.9	不透明度0.9	不透明度1

消えるボールの不透明度を1未満（半透明）に設定する

消えないボールの不透明度は1（完全に不透明）

ボールを徐々に消す

本書では、ボールの表示をだんだん薄くして消すために、ボールごとに不透明度の情報を持

Fig. 5-24 消えるボールを消す

ボールの配列

＜要素0＞ 種類2 位置1 座標(17,2) 不透明度1	＜要素1＞ 種類4 位置2 座標(16,2) 不透明度0.9	＜要素2＞ 種類4 位置3 座標(15,2) 不透明度0.9	＜要素3＞ 種類4 位置4 座標(14,2) 不透明度0.9	＜要素4＞ 種類3 位置5 座標(13,2) 不透明度1

消えるボール
(不透明度が1未満)

消えないボール
(不透明度が1)

＜要素0＞ 種類2 位置1 座標(17,2) 不透明度1	＜要素1＞ 種類4 位置2 座標(16,2) 不透明度0.8	＜要素2＞ 種類4 位置3 座標(15,2) 不透明度0.8	＜要素3＞ 種類4 位置4 座標(14,2) 不透明度0.8	＜要素4＞ 種類3 位置5 座標(13,2) 不透明度1

消えるボールの不透明度を
時間とともに減少させる

＜要素0＞ 種類2 位置1 座標(17,2) 不透明度1	＜要素1＞ 種類3 位置5 座標(13,2) 不透明度1

不透明度が0以下になったら
消えるボールを削除して
後方の要素を前方に詰める

っています。消えないボールは不透明度が「1(完全に不透明)」ですが、消えるボールは不透明度を「1未満(半透明)」に設定します。

消えるボールについては、時間とともに不透明度を減少させます(Fig. 5-24)。この不透明度に応じて、ボタンを半透明で描画すれば、ボールがだんだん薄くなって消えていくような効果が得られます。

不透明度が0以下になったら、ボールが見えなくなるので、ボールを完全に削除します。ボールの配列について、消えるボールの後方にある要素を前方に動かすことによって、配列からボールを取り除きます。

ボールを削除してできる空いた場所は、前方のボールが後退することによって埋めます。これは「軌道に沿って進むボール」(→p. 278)で解説したように、ボールの間隔を調整することによって行います。

プログラム

P

List 5-5は軌道上に並んだボールを消すプログラムです。ステージの移動処理と、ボールを削除する処理を掲載しました。

移動処理(Move関数)では、軌道上で同じ種類のボールが並んでいるかどうかを調べます。規定数(3個)以上のボールが並んでいたら、ボールを消します。ボールの不透明度を1未満(0.9)に設定して、消えるボールにします。

消えるボール(不透明度が1未満のボール)については、時間とともに不透明度を少しずつ減少させます。不透明度が0以下になったら、ボールを削除する処理(EraseBall関数)を呼び出します。ここでは、消えるボールをボールの配列から除去することによって、完全に削除します。

List 5-5 軌道上に並んだボールを消す（CBallOnRailStageクラス）

```
// 移動処理
bool CBallOnRailStage::Move(const CInputState* is) {

    // ...（中略）...

    // ボールの座標を更新する
    for (int i=0; i<BallCount; i++) {

        // 軌道上の位置からボールの座標を計算する
        SetBallXY(i);

        // 消えるボール（不透明度が1未満のボール）を、
        // 時間とともに少しずつ消す
        float &a=Ball[i].Alpha;
        if (a<1) {

            // 不透明度を減少させる
            a-=0.1f;

            // 完全に透明になったら、
            // ボールを削除する
            if (a<=0) {
                EraseBall(i);
                i--;
            }
        }
    }

    // ...（中略）...

    // 軌道上で規定数（3個）以上のボールが並んだら、ボールを消す
    for (int i=0; i<BallCount; i++) {

        // 同じ種類のボールが規定数以上並んでいるかどうかを調べる
        int type=Ball[i].Type;
        int j;
        for (
            j=i+1;
            j<BallCount &&

            // 同じ種類のボールである
            Ball[j].Type==type &&

            // ボール間の距離がボール1個分である
            Ball[j].Pos==Ball[j-1].Pos+1;

            j++
        ) ;
```

```
            // 規定数以上並んでいたら、ボールを消す
            if (j-i>=BALL_ON_RAIL_ERASE) {
                for (; i<j; i++) {

                    // ボールの不透明度を1未満にする
                    Ball[i].Alpha=min(0.9f, Ball[i].Alpha);
                }
            }
        }

        return true;
    }

// ボールを削除する処理
void CBallOnRailStage::EraseBall(int index) {

    // ボールの数を減らす
    BallCount--;

    // 配列上で削除するボールよりも後のボールを、
    // 1個ずつ前方にずらす
    for (int i=index; i<BallCount; i++) {
        Ball[i]=Ball[i+1];
    }
}
```

ぶら下がったボール

ステージ上方に多数のボールがぶら下がっている状態です。ボールが互い違いに結合しているため、ぶどうの房がぶら下がっているようにも見えます。

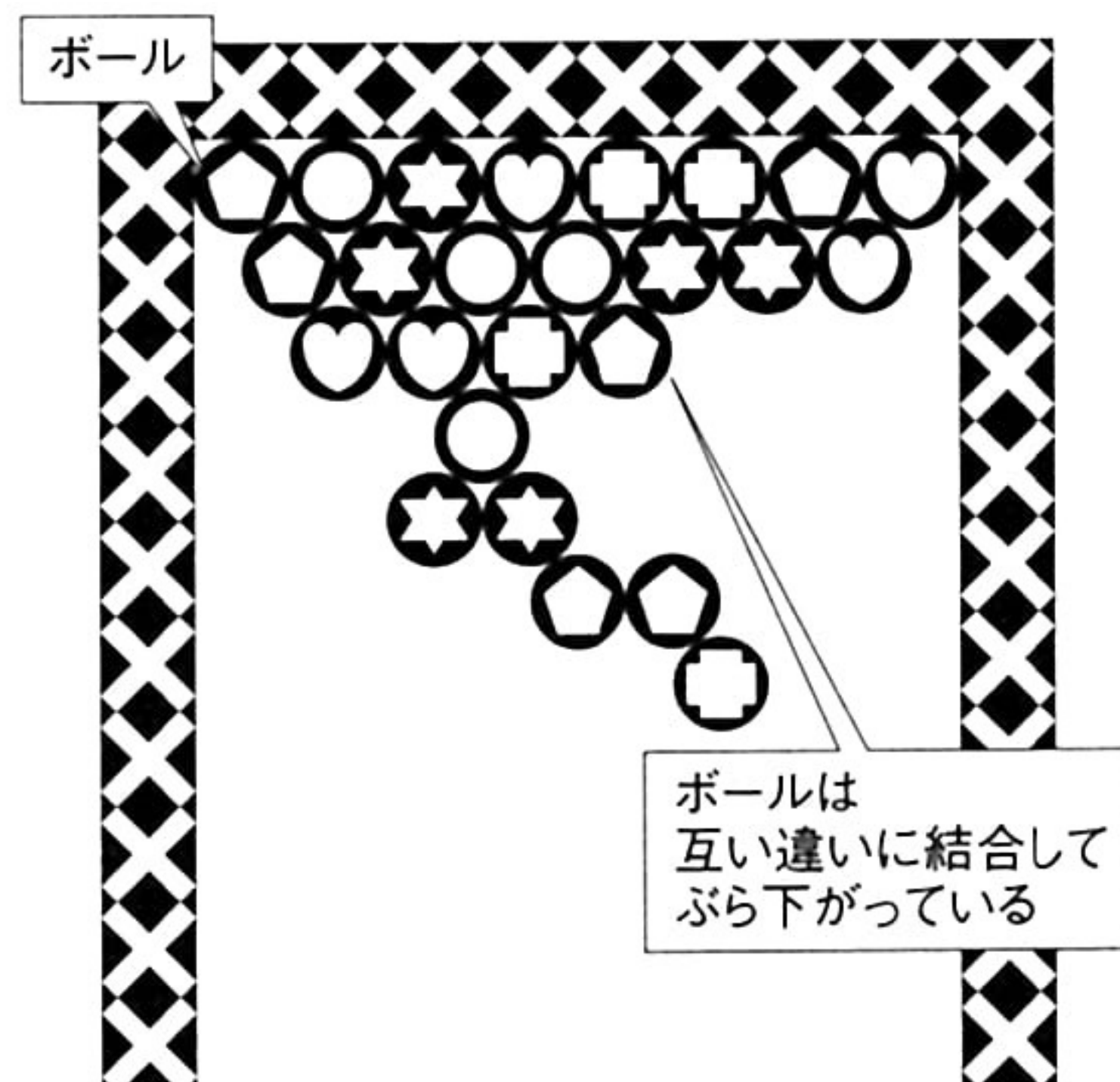
ステージ上方に、ボールが互い違いに結合してぶら下がっています（Fig. 5-25）。ボールは格子状ではなく、六角形状に並んでいます。

ぶら下がったボールは『パズルボブル』に採用されています。このゲームでは、ぶら下がったボールに対して、新しいボールを撃ち込んで、貼り付けることができます。上手にボールを貼り付けて、同じ種類のボールを規定数以上隣接させると、ボールを消すことができます。

また、『戦球』はボールを積み上げるゲームですが、ぶら下がったボールと同様に、ボールが六角形状に並びます。ボールが互い違いに積まれるために生じた段差を、着地したボールが転がり落ちるので、見ていて楽しいユニークな動きのゲームになっています。

Fig. 5-25 ぶら下がったボール

ボール

ボールは
互い違いに結合して
ぶら下がっている

アルゴリズム

ぶら下がったボールを実現するには、ステージやボールをセルで表現します。ただし、ボールを格子状ではなく、互い違いの六角形状に配置するので、少し工夫が必要です。

そこで、ステージのセルとボールのセルを分けて考えます。ステージのセルは「ステージセル」、ボールのセルは「ボールセル」と呼ぶことにしましょう。

ステージセルに関しては、ステージの壁を「=」で表しました(Fig. 5-26)。壁は格子状に配置するので、ステージセルに関しては特別な工夫は必要ありません。

ボールセルに関しては、ボールを「0〜4」の数字で、ボールを配置できない場所を「+」で表しました(Fig. 5-27)。ボールを画面に表示するときには、ボールが互い違いになるように、奇数段(奇数番目の段)と偶数段(偶数番目の段)をずらして描画します。そこで、画面のイメージに近くなるように、ボールセルもずらして示すことにします(Fig. 5-28)。

Fig. 5-26 ステージセル

壁
(=)

=	=	=	=	=	=	=	=	=	=
=									=
=									=
=									=
=									=
=									=
=									=
=									=
=									=
=									=
=									=

Fig. 5-27　ボールセル

ボールを配置できない場所（＋）

ボール（0〜4）

+	+	+	+	+	+	+	+	+	+
+	4	0	1	2	3	3	4	2	+
+	4	1	0	0	1	1	2	+	+
+		2	2	3	4				+
+			0					+	+
+			1	1					+
+				4	4			+	+
+						3			+
+								+	+
+									+
+	+	+	+	+	+	+	+	+	

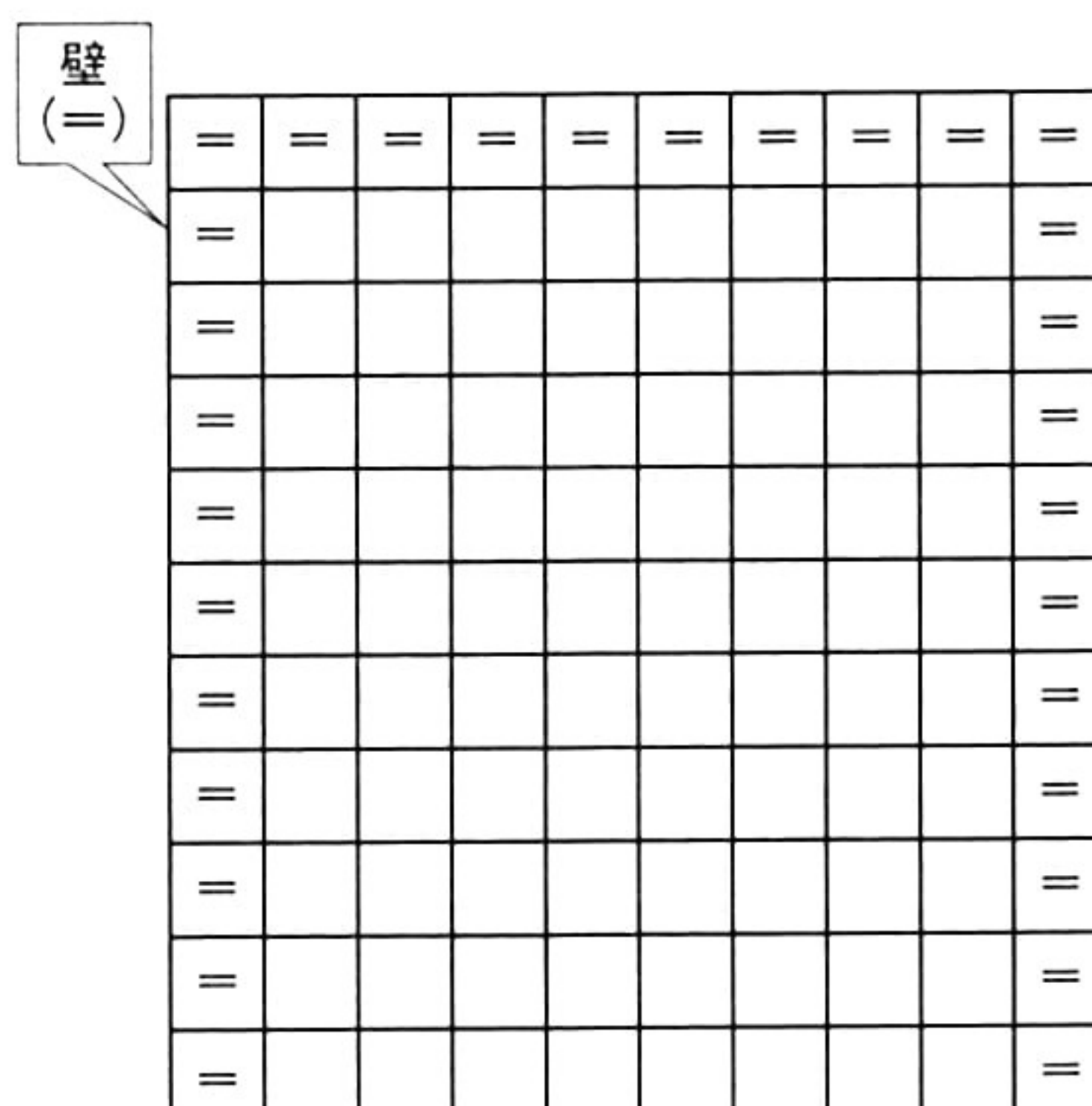
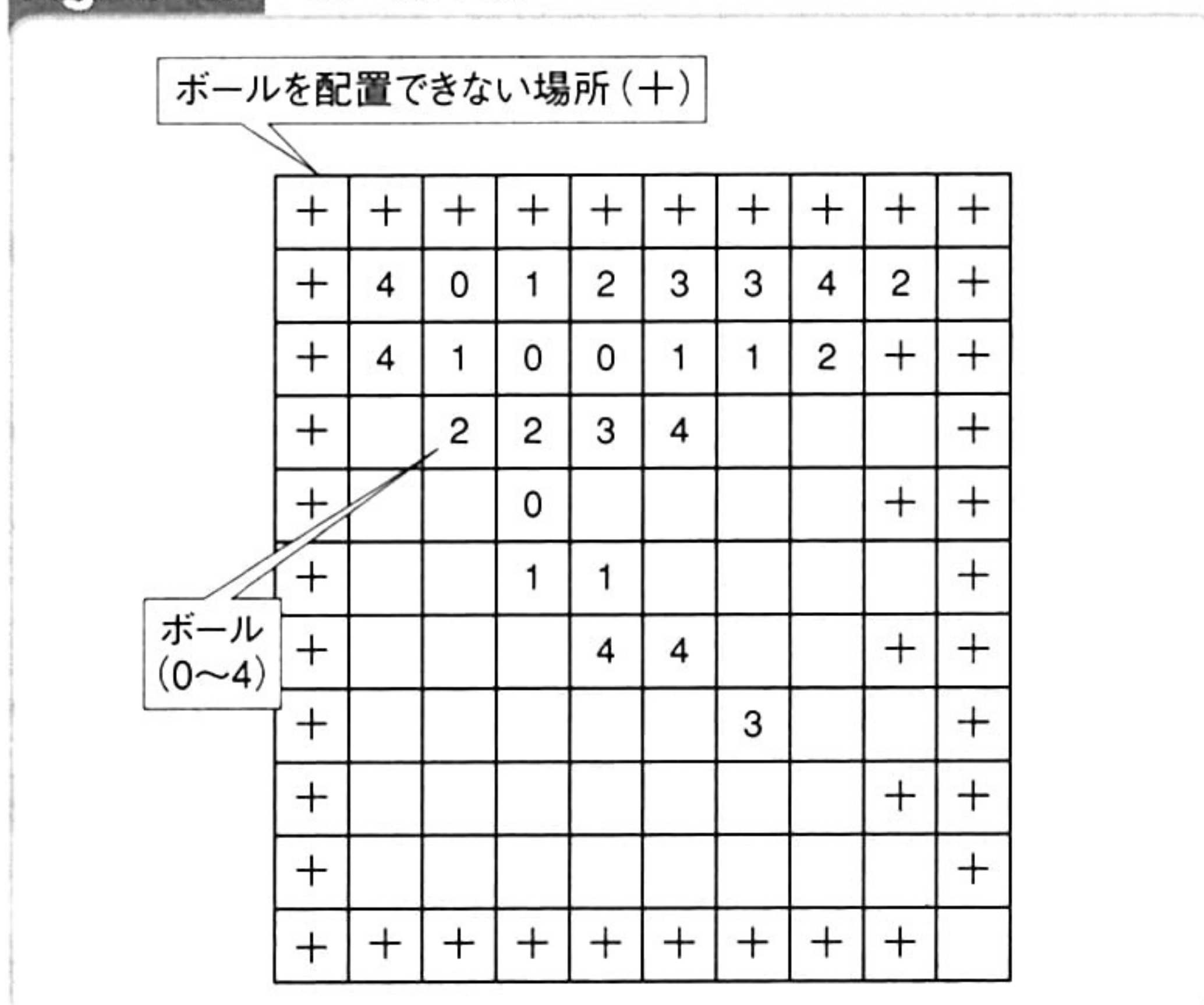
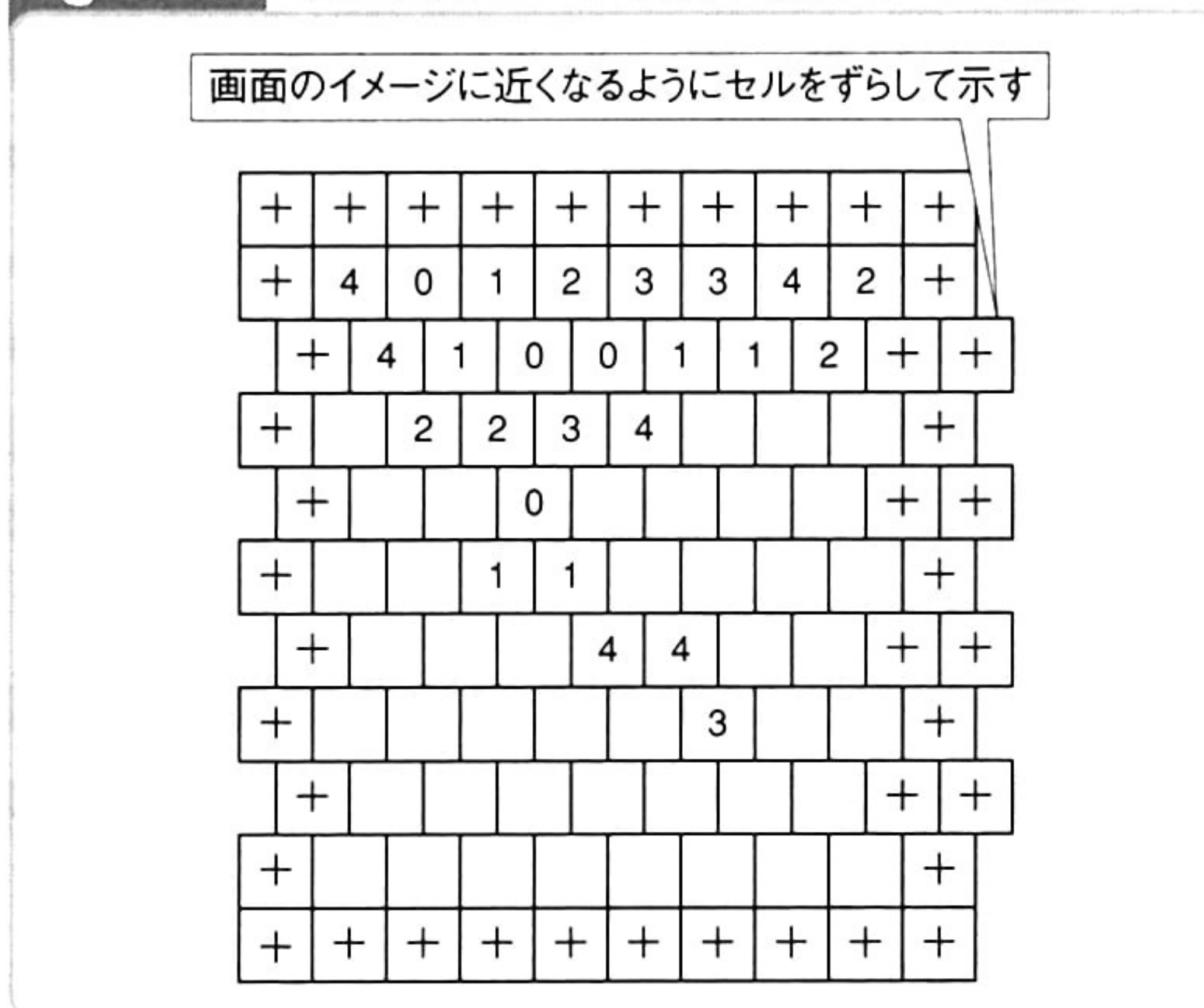



Fig. 5-28　画面のイメージに近づけたボールセル

画面のイメージに近くなるようにセルをずらして示す

+	+	+	+	+	+	+	+	+	+
+	4	0	1	2	3	3	4	2	+
+	4	1	0	0	1	1	2	+	+
+		2	2	3	4				+
+			0					+	+
+			1	1					+
+				4	4			+	+
+						3			+
+								+	+
+									+
+	+	+	+	+	+	+	+	+	+

ボールの座標

ステージを描画するには、ステージセルを調べて、セルの種類に応じたグラフィックを表示します。ボールを描画するときも同様に、ボールセルを調べて、セルの種類に対応したグラフィックを表示します。ただし、ボールを互い違いに配置するために、ボールの座標を特別な方法で計算します。

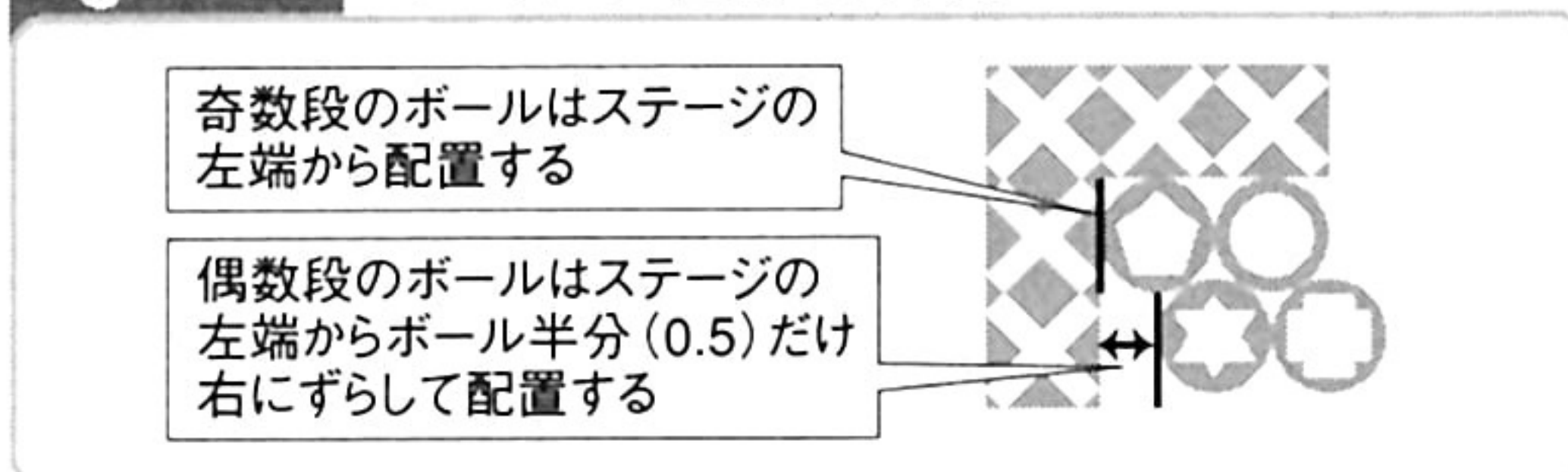
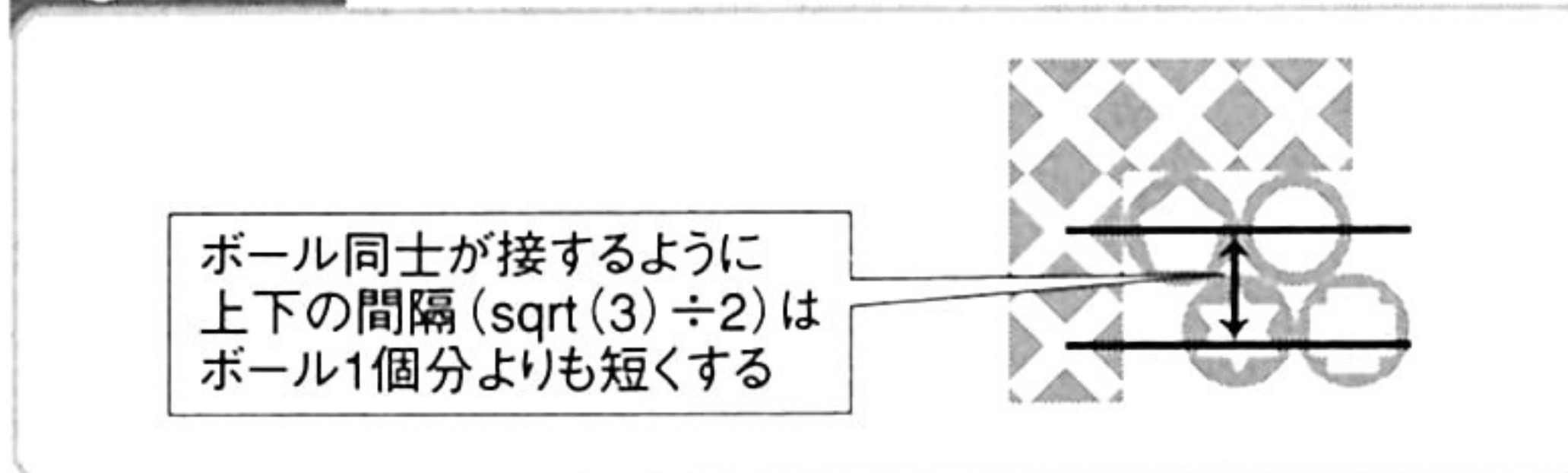
ボールのX座標は、奇数段か偶数段かによって、計算方法が変わります（Fig. 5-29）。奇数段（1、3、5、7…段目）のボールは、ステージの左端から配置します。偶数段（2、4、6、8…段目）のボールは、ステージの左端から、ボール半分だけ右にずらした位置から配置します。ボールの直径を「1」とすると、偶数段のボールは「0.5」ずつ右にずらします。

ボールのY座標は、奇数段も偶数段も計算方法は同じです（Fig. 5-30）。隣接したボール同士が接するように、ボール1個分よりも短い間隔で、ボールを上下に並べます。ボールの直径を「1」とすると、上下の間隔は「sqrt（3）÷2」になります。

Fig. 5-29　ボールのX座標を計算する

奇数段のボールはステージの左端から配置する

偶数段のボールはステージの左端からボール半分（0.5）だけ右にずらして配置する

Fig. 5-30　ボールのY座標を計算する

ボール同士が接するように上下の間隔（sqrt（3）÷2）はボール1個分よりも短くする

プログラム

List 5-6はぶら下がったボールのプログラムです。ステージの初期化処理と描画処理、セルがボールかどうかを判定する処理、ボールの中心座標を求める処理を掲載しました。

初期化処理（Init関数）では、ステージセルとボールセルを初期化します。ボールは互い違いに配置するため、ボールセルの奇数段と偶数段では、ボールの数が異なっています。

描画処理（Draw関数）では、ぶら下がったボールを描画します。ボールセルを調べて、セルの種類に対応したグラフィックを表示します。ボールを互い違いに配置するために、ボールの中心座標を求める処理（GetBallX関数、GetBallY関数）を使います。

ボールの中心のX座標を求める処理（GetBallX関数）では、指定したセル座標のボールについて、描画座標のX座標を計算します。奇数段のボールに対して、偶数段のボールは右にボール半分だけずらします。

ボールの中心のY座標を求める処理（GetBallY関数）では、指定したセル座標のボールについて、描画座標のY座標を計算します。ボールの上下の間隔が、隣接したボール同士が接する距離になるようにします。

描画処理では、消えるボールをだんだん薄くなるように描画する処理も行います。消えるボールの詳細は「ぶら下がった同じ種類のボールを消す」（→p. 315）で解説します。

セルがボールかどうかを判定する処理（IsBall関数）は、セルがボールのときにtrueを返します。いくつかの処理でセルがボールかどうかを判定する必要があるので、関数にまとめました。

List 5-6　ぶら下がったボール（CHangingBallStageクラス）

```
// 初期化処理
void CHangingBallStage::Init() {

    // ステージセルの初期化
    StageCell->Init(
        "                "
        "================"
        "=              ="
        "=              ="
        "=              ="
        "=              ="
        "=              ="
        "=              ="
        "=              ="
        "=              ="
        "=              ="
        "=              ="
    );

    // ボールセルの初期化
    BallCell->Init(
        "++++++++++++++++"
        "+01234012340123+"
        "+1234012340123++"
        "+              +"
        "+             ++"
        "+              +"
```

```
        "+              ++"
        "+               +"
        "+              ++"
        "+               +"
        "+              ++"
        "++++++++++++++++"
    );

    // ...(中略)...
}

// 描画処理
void CHangingBallStage::Draw() {

    // ...(中略)...

    // ぶら下がったボールの描画
    for (int y=0; y<bys; y++) {
        for (int x=0; x<bxs; x++) {
            char c=BallCell->Get(x, y);

            // セルがボールの場合には、ボールを描画する
            if (IsBall(c&0x3f)) {

                // 通常のボールは黒色で描画する
                D3DCOLOR color=COL_BLACK;

                // 消えるボールはだんだん薄くなるように描画する
                if (c&0x40) {
                    float f=(float)Time/30;
                    color=D3DXCOLOR(f, f, f, 1);
                }

                // ボールの種類に応じたグラフィックを描画する
                Game->Texture[TEX_ORB0+(c&0x3f)-'0']->Draw(
                    (GetBallX(x, y)-0.5f)*sw,
                    (GetBallY(y)-0.5f)*sh,
                    sw, sh, 0, 0, 1, 1, color);
            }
        }
    }

    // ...(中略)...
}

// セルがボールかどうかを判定する処理
bool CHangingBallStage::IsBall(char c) {
    return '0'<=c && c<'0'+HANGING_BALL_TYPE;
}
```

```
// ボールの中心のX座標を求める処理
// セル座標から描画座標を求める
float CHangingBallStage::GetBallX(int cx, int cy) {
    return cx+(cy+1)%2*0.5f+0.5f;
}

// ボールの中心のY座標を求める処理
// セル座標から描画座標を求める
float CHangingBallStage::GetBallY(int cy) {
    return (cy-1)*sqrtf(3)*0.5f+2.5f;
}
```

SAMPLE

「HANGING BALL」は「ぶら下がったボール」「撃ったボールが跳ね返る」「撃ったボールがぶら下がる」「ぶら下がった同じ種類のボールを消す」のサンプルです。

レバーの左右（カーソルキーの左右）で砲台が左右に回転します。ボタン0（Zキー）を押すと、ボールを撃つことができます。

撃ったボールは、砲台が向いている方向に飛んでいきます。壁に当たると、ボールは跳ね返ります。ステージの下端から出ると、ボールは消えて、砲台上に新しいボールが出現します。

撃ったボールがぶら下がったボールに接触すると、撃ったボールがぶら下がります。上手にボールをぶら下げて、同じ種類のボールを規定数（3個）以上隣接させると、消すことができます。

HANGING BALL → p. 389

撃ったボールが跳ね返る

「ぶら下がったボール」（→p. 301）において、レバー操作で砲台を回転させて、任意の方向にボールを撃つアクションです。撃ったボールは壁に当たると跳ね返ります。跳ね返りを上手に利用すると、ぶら下がったボールの奥まった場所に、ボールを撃ち込むことができます。

砲台はステージの下方に配置されています（Fig. 5-31）。レバーを左右に入力すると、砲台が左右に回転します。

ボタンを押すと、ボールを撃つことができます（Fig. 5-32）。ボールは砲台が向いている方向に飛んでいきます。ボールはまっすぐに飛んでいきますが、壁に当たると跳ね返ります（Fig. 5-33）。右の壁に当たると左に、左の壁に当たると右に、上の壁に当たると下に反発します。ステージの下端から出ると、ボールは消えます。

Fig. 5-31 砲台を回転させる

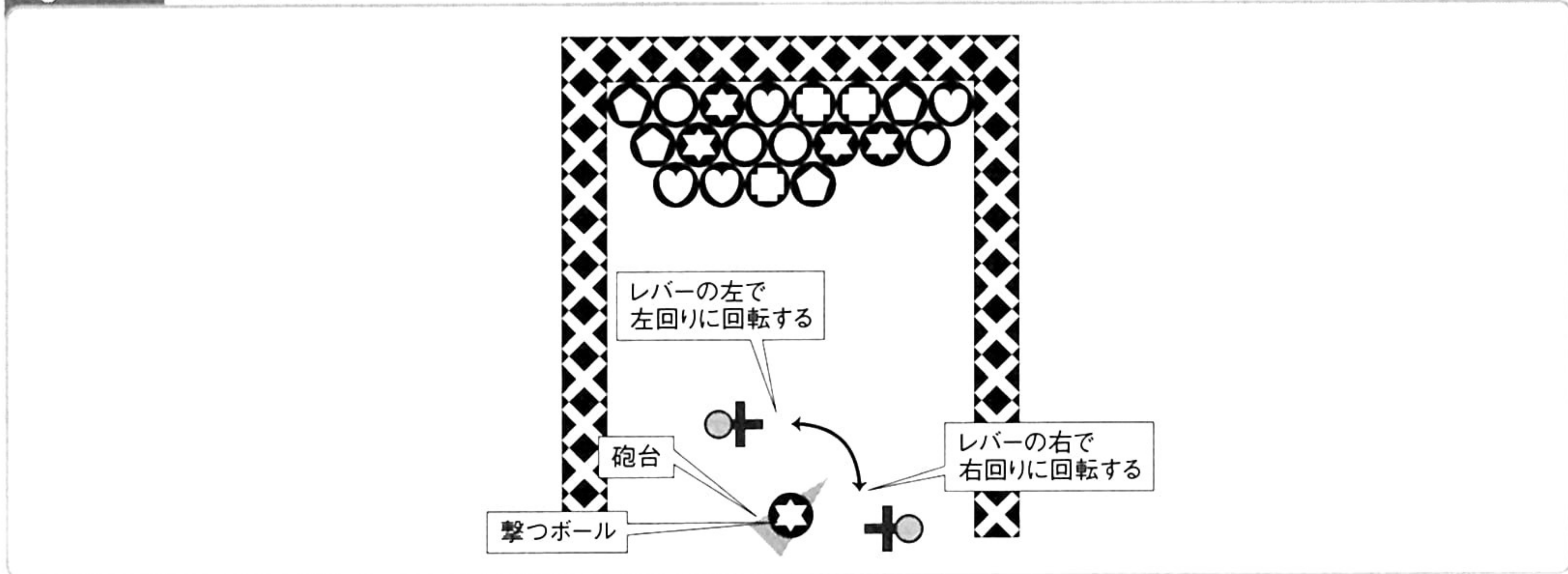



Fig. 5-32 ボールを撃つ

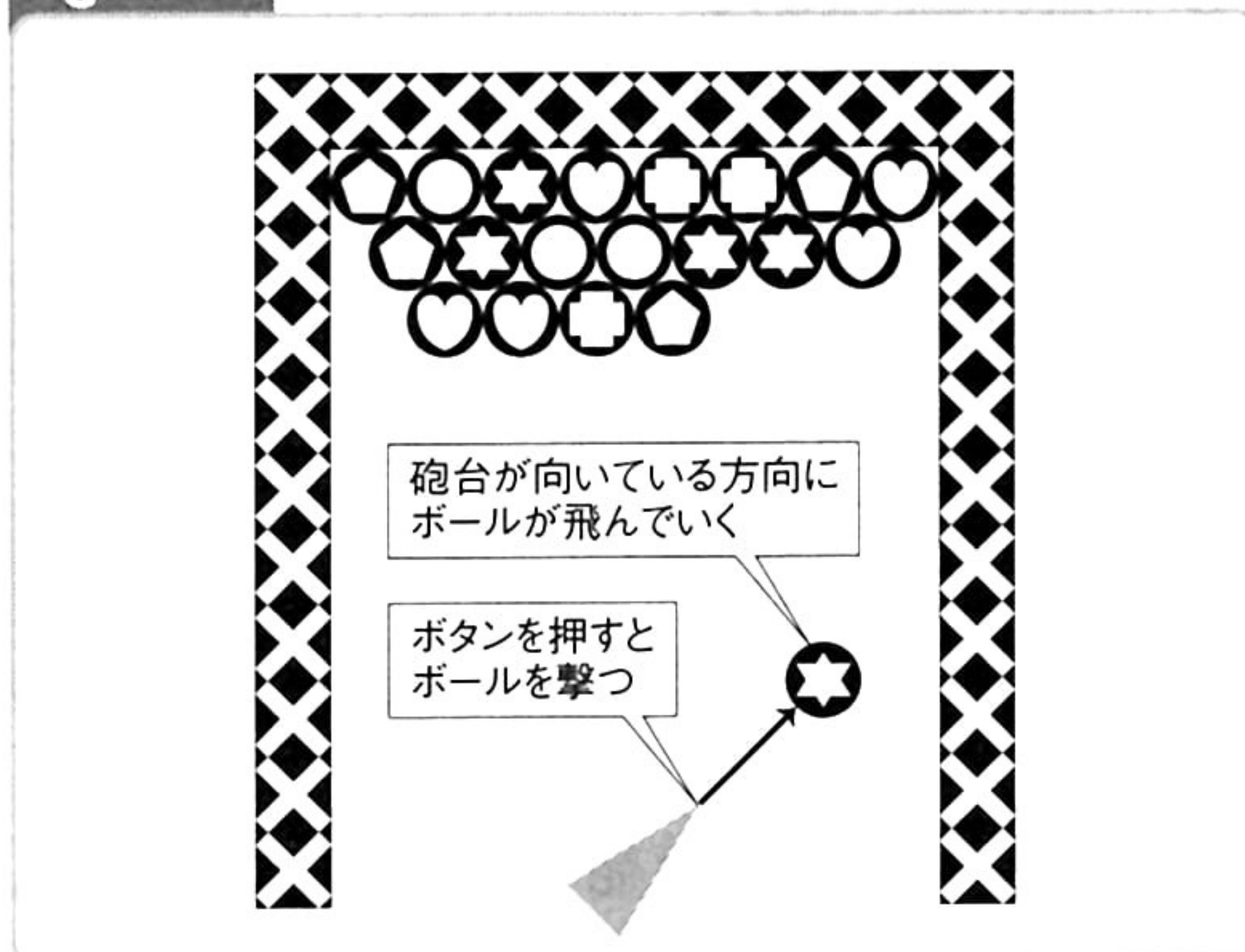
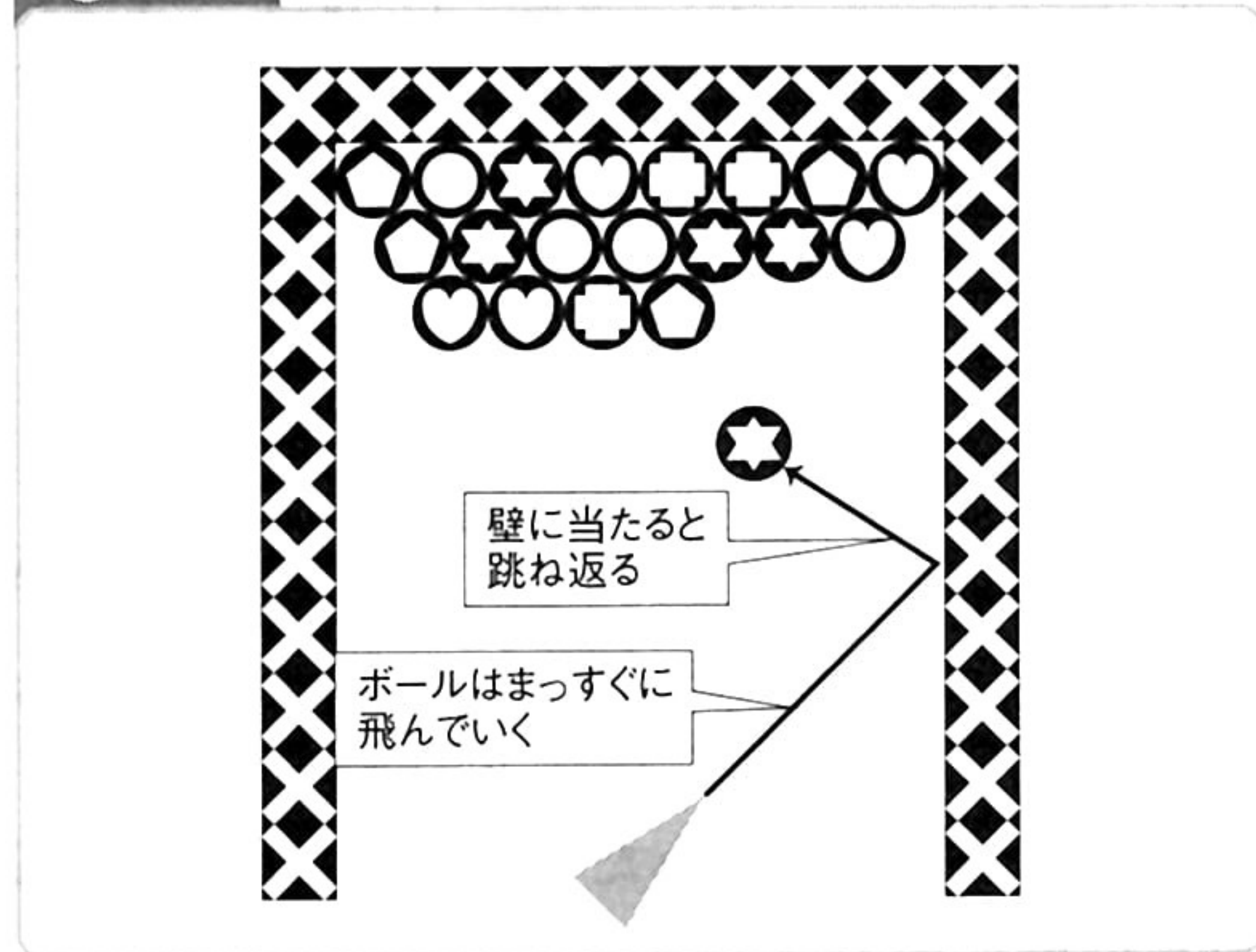



Fig. 5-33 ボールが跳ね返る

アルゴリズム

砲台を回転させて、砲台の方向にボールを撃つ方法は、「ボールを任意の方向に撃つ」(→p. 290)と同じです。レバーの入力方向に応じて、砲台の角度を変化させ、砲台を左右に回転させます(Fig. 5-34)。

ボタンを押したら、ボールを撃ちます(Fig. 5-35)。ボールを砲台が向いている方向に飛ばすために、砲台の角度を使って、ボールの速度を計算します。砲台の角度(0.0〜1.0)をSDir、ボールの速さをBSpeed、円周率をPIとすると、ボールの速度(BVX, BVY)は、

BVX=cos (SDir×PI×2)×BSpeed
BVY=sin (SDir×PI×2)×BSpeed

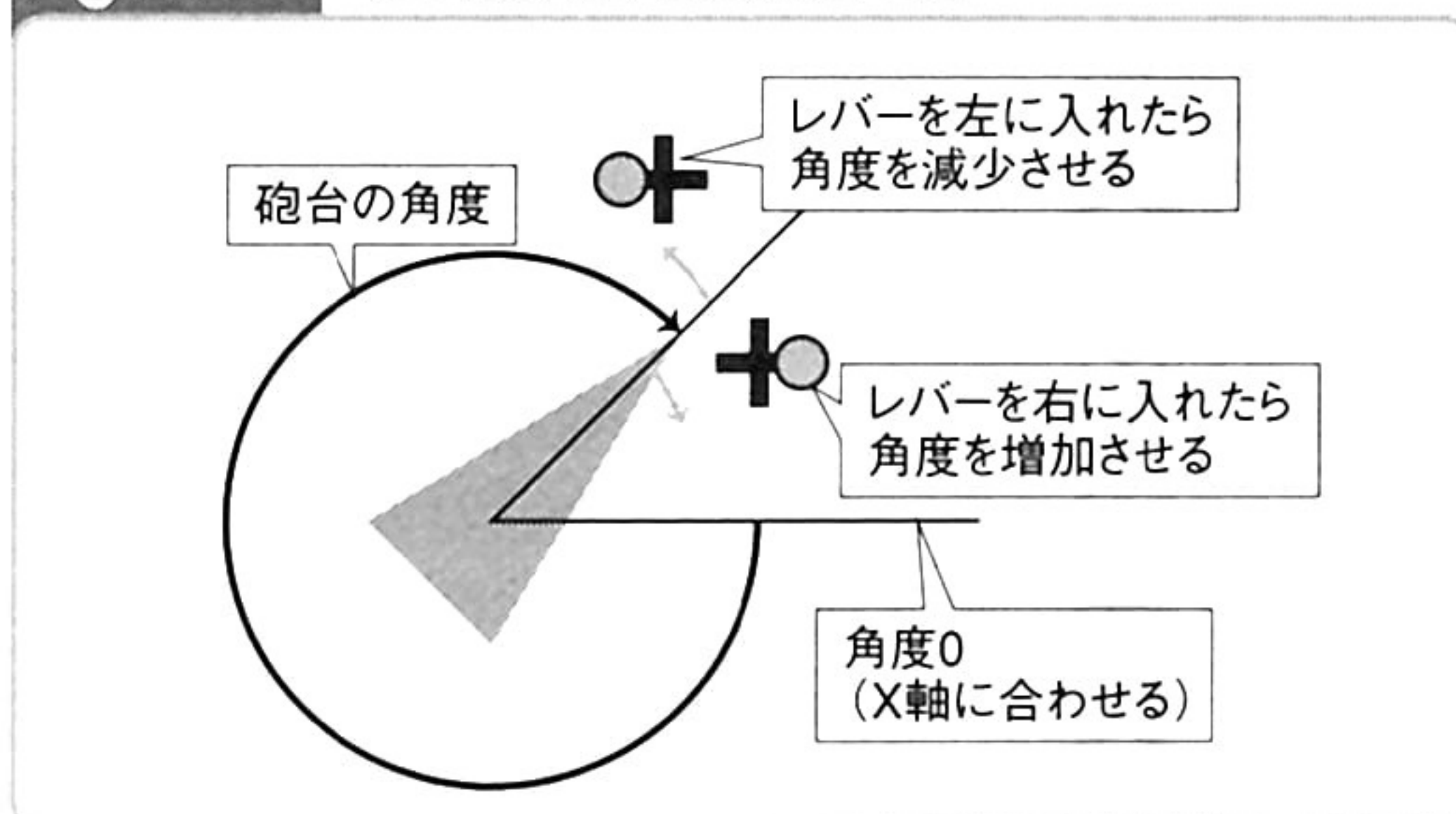
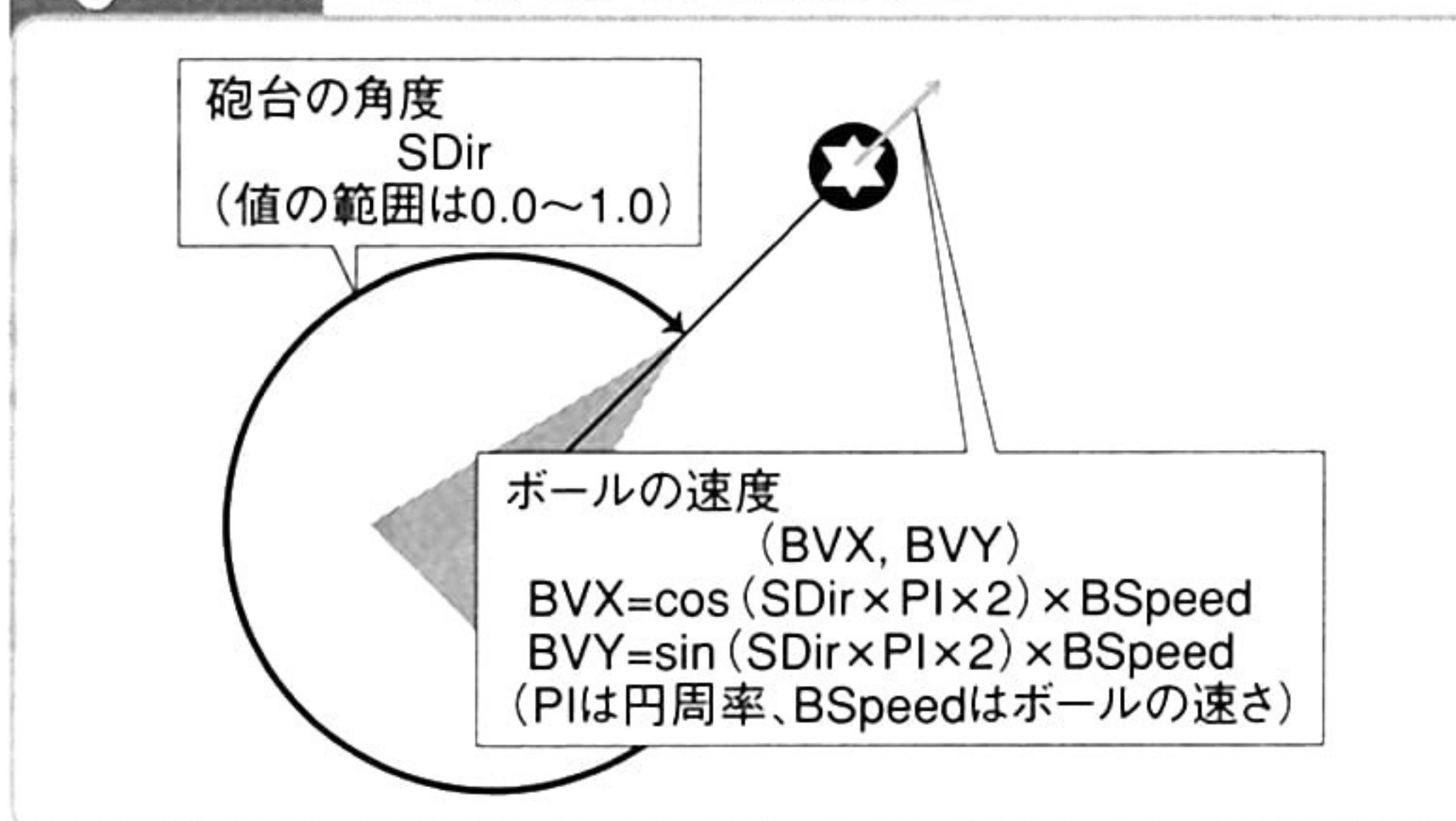
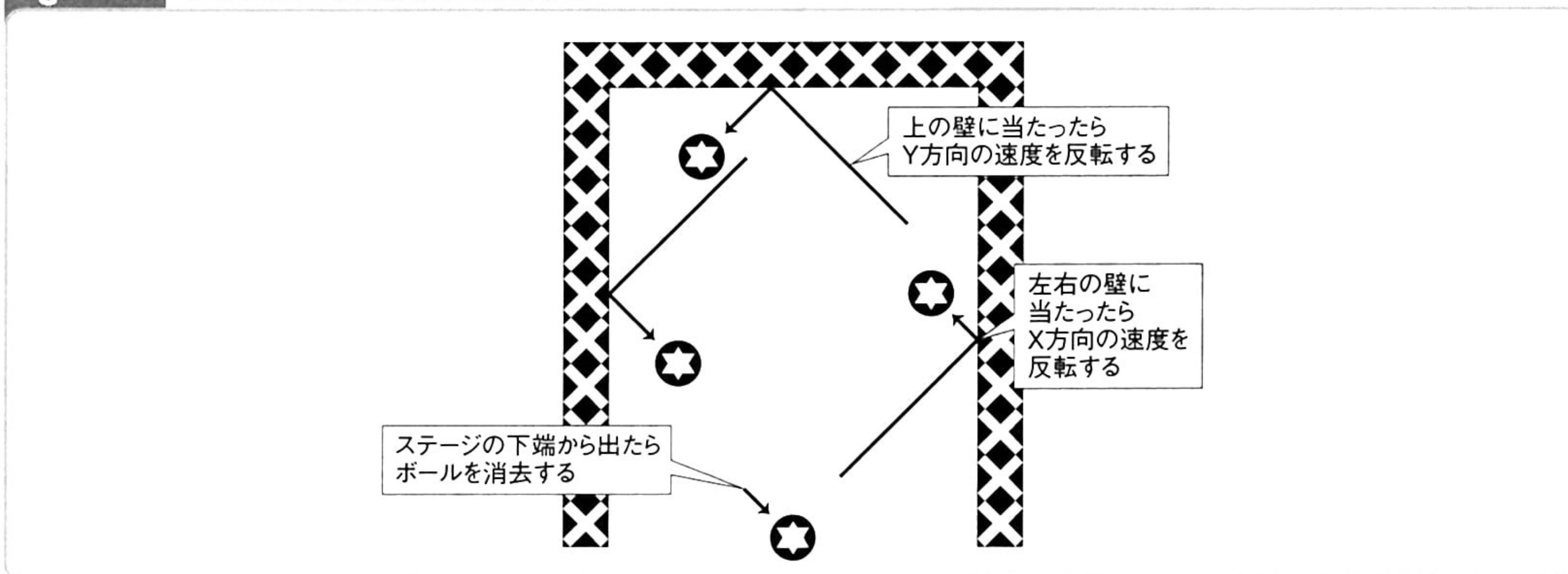
となります。

Fig. 5-34 砲台の角度を変化させる

砲台の角度
レバーを左に入れたら
角度を減少させる
レバーを右に入れたら
角度を増加させる
角度0
（X軸に合わせる）

Fig. 5-35 ボールの速度を計算する

砲台の角度
SDir
（値の範囲は0.0〜1.0）
ボールの速度
（BVX, BVY）
BVX=cos（SDir×PI×2）×BSpeed
BVY=sin（SDir×PI×2）×BSpeed
（PIは円周率、BSpeedはボールの速さ）

Fig. 5-36 壁に当たったら速度を反転する

上の壁に当たったら
Y方向の速度を反転する
左右の壁に
当たったら
X方向の速度を
反転する
ステージの下端から出たら
ボールを消去する

ボールが壁に当たったら、速度を反転することによって、ボールを反発させます（Fig. 5-36）。左右の壁に当たったときにはX方向の速度を、上の壁に当たったときにはY方向の速度を反転します。ステージの下端から出たときには、ボールを消去して、砲台上に新しいボールを生成します。

プログラム

List 5-7は撃ったボールが跳ね返るプログラムです。ステージの移動処理と描画処理、撃ったボールを移動する処理を掲載しました。

移動処理（Move関数）は、入力状態と射出状態に分かれています。入力状態では、レバー入力に応じて、砲台を左右に回転させます。ボタンを押したら、ボールを飛ばすために、射出状態に移行します。ボールの速度は砲台の角度から計算します。

射出状態では、撃ったボールを移動する処理（MoveBall関数）を呼び出します。ここでは、ボールの座標に速度を加算することによって、ボールを直線的に動かします。ボールが左右または上の壁に当たったら、速度を反転させて、ボールを反発させます。ボールがステージの下

端から出たら、初期状態に移行します。

描画処理（Draw関数）では、砲台と撃ったボールを描画します。砲台については、三角形を砲台の角度に合わせて回転させたものを描きます。撃ったボールについては、ボールの種類に応じたグラフィックを表示します。

List 5-7　撃ったボールが跳ね返る（CHangingBallStageクラス）

```
// 移動処理
bool CHangingBallStage::Move(const CInputState* is) {

    // ...（中略）...

    // 入力状態
    if (State==1) {

        // レバーを左右に入力したら、砲台を左右に回転させる
        if (is->Left && SDir>0.52f) SDir-=0.005f;
        if (is->Right && SDir<0.98f) SDir+=0.005f;

        // 砲台の角度から、撃つボールの速度を計算する
        float rad=SDir*D3DX_PI*2;
        BVX=cosf(rad)*0.5f;
        BVY=sinf(rad)*0.5f;

        // ボタンを押したら、射出状態に移行する
        if (!PrevButton && is->Button[0]) {
            State=2;
        }
        PrevButton=is->Button[0];
    }

    // 射出状態
    if (State==2) {

        // 撃ったボールを移動する
        MoveBall();

        // ...（中略）...
    }

    // ...（中略）...
}

// 撃ったボールを移動する処理
void CHangingBallStage::MoveBall() {

    // ボールの座標を更新する
    BX+=BVX;
    BY+=BVY;
```

```
        // ボールが左右の壁に当たったら、
        // X方向の速度を反転させる
        if (BX<1.5f || BX>=MAX_X-1.5f) {
            BVX=-BVX;
        }

        // ボールが上の壁に当たったら、
        // Y方向の速度を反転させる
        if (BY<2.5f) {
            BVY=-BVY;
        }

        // ボールがステージの下端から出たら、
        // 初期状態に移行する
        if (BY>=MAX_Y+0.5f) {
            State=0;
        }
}

// 描画処理
void CHangingBallStage::Draw() {

    // ...（中略）...

    // 砲台の描画
    // 砲台の方向に合わせて、三角形を描画する
    float
        rad=SDir*D3DX_PI*2,
        c=cosf(rad), s=sinf(rad),
        f0=2.0f, f1=1.0f, f2=0.5f;
    Game->Texture[TEX_FILL]->Draw(
        (SX+c*f0)*sw, (SY+s*f0)*sh, COL_LGRAY, 0, 0,
        (SX+c*f0)*sw, (SY+s*f0)*sh, COL_LGRAY, 0, 1,
        (SX-c*f1+s*f2)*sw, (SY-s*f1-c*f2)*sh, COL_LGRAY, 1, 0,
        (SX-c*f1-s*f2)*sw, (SY-s*f1+c*f2)*sh, COL_LGRAY, 1, 1
    );

    // 撃ったボールの描画
    if (State==1 || State==2) {
        Game->Texture[TEX_ORB0+BType]->Draw(
            (BX-0.5f)*sw, (BY-0.5f)*sh, sw, sh,
            0, 0, 1, 1, COL_BLACK);
    }
}
```

撃ったボールがぶら下がる

「ぶら下がったボール」(→p. 301)において、砲台から撃ったボールがぶら下がるアクションです。ボールを適切な位置にぶら下げることにより、同じ種類のボールを隣接させて消すことが、ゲームの目的です。

砲台から撃ったボールが、ぶら下がったボールに接触すると、ボールがくっついて、その場所にぶら下がります(Fig. 5-37)。接触したときのボールの位置関係によって、撃ったボールがぶら下がる場所が変わります(Fig. 5-38)。

微妙な位置の違いによって、ぶら下がる場所が変わるところが、このアクションの面白いところです。思ったとおりの場所にぶら下げるには、砲台の角度を細かく調整して、目的の場所を狙う必要があります。

Fig. 5-37 撃ったボールがぶら下がる

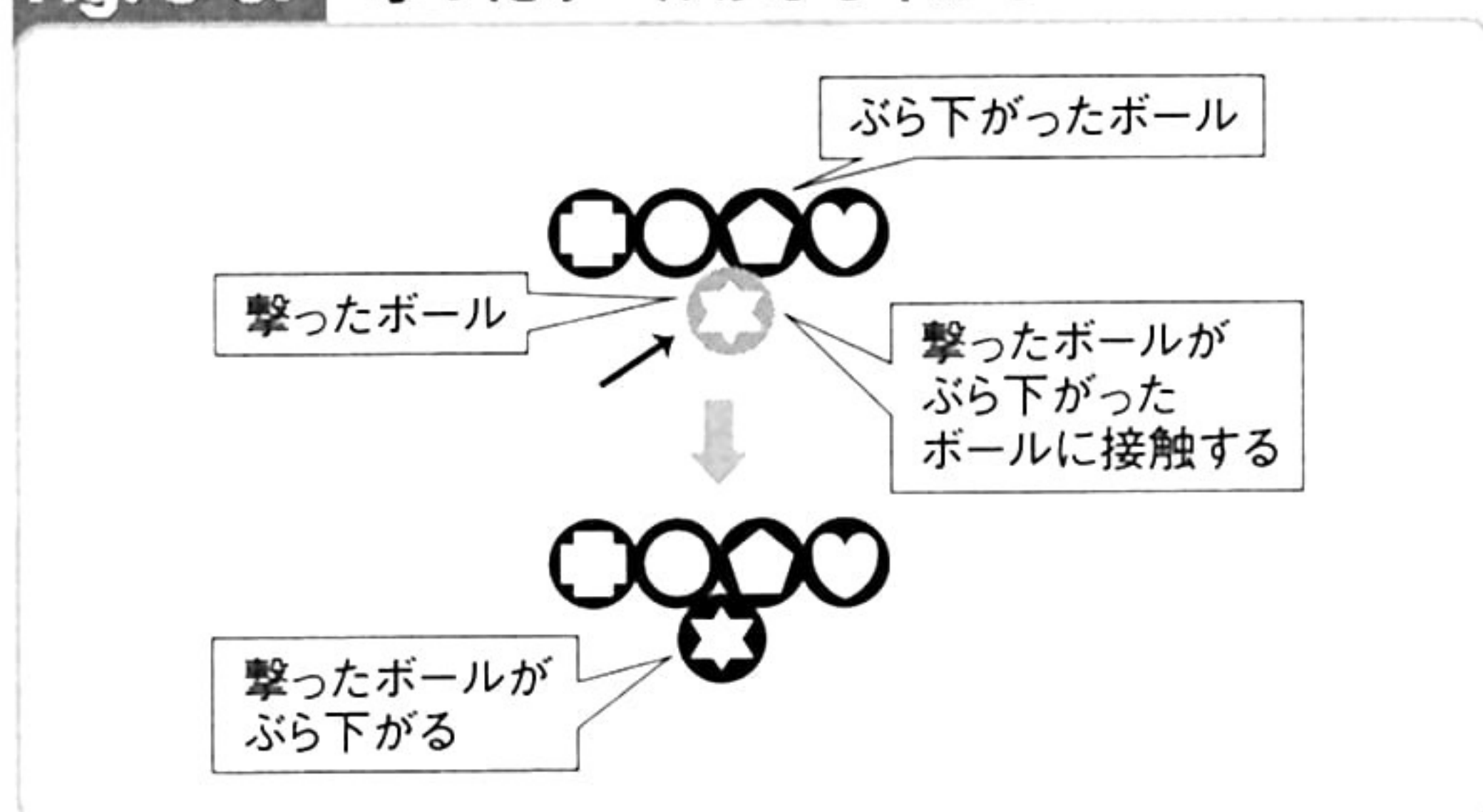
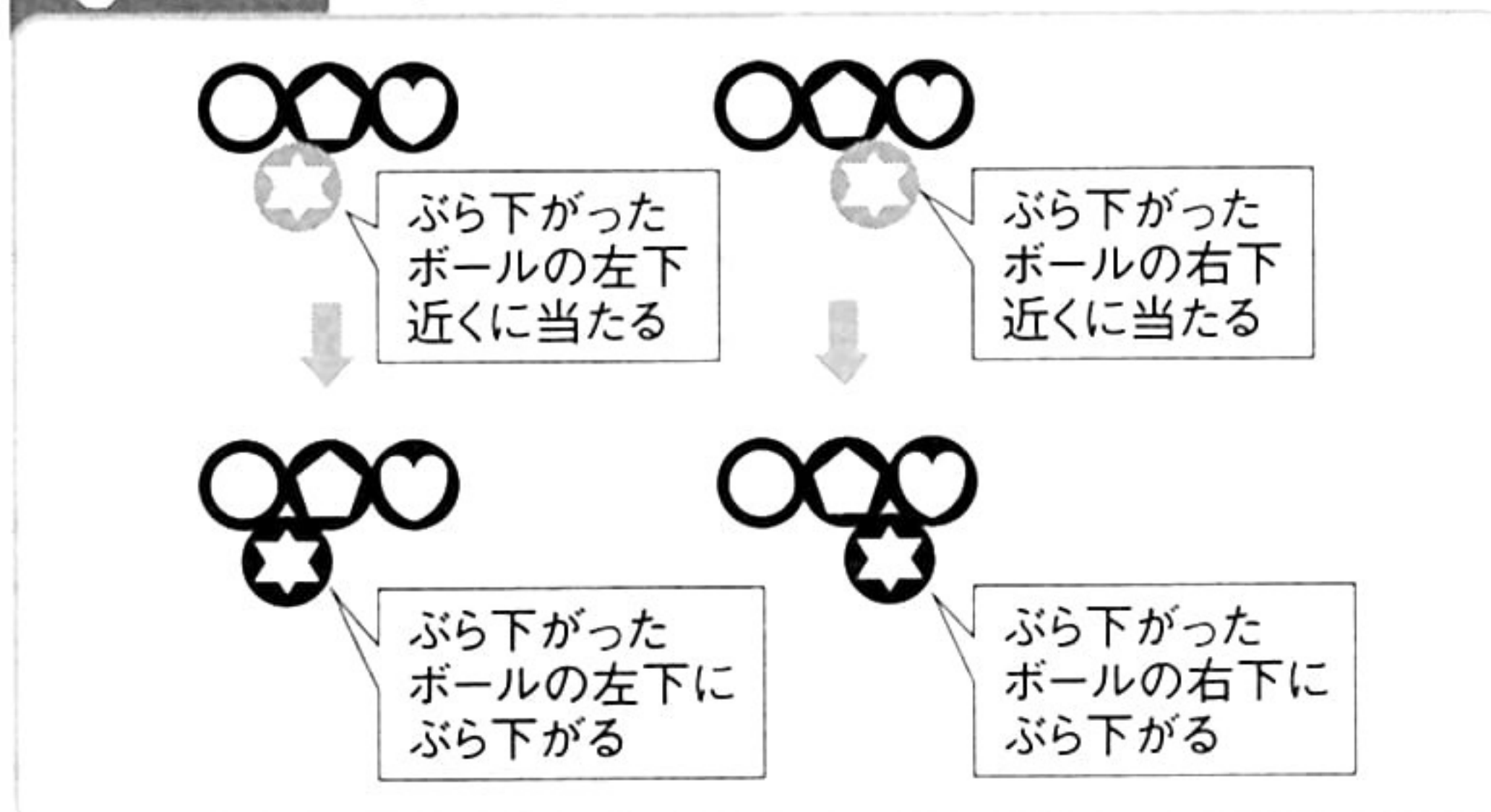



Fig. 5-38 当たる位置によってぶら下がる場所が変わる

アルゴリズム

撃ったボールをぶら下げるには、撃ったボールと、ぶら下がったボールとの間で、当たり判定処理を行います(Fig. 5-39)。ボール同士の当たり判定処理なので、ボールの中心間の距離が一定値よりも小さくなったら、接触したとみなせます。

本書のサンプルでは、ボールの中心間の距離が、ボールの直径の80%未満になったら、当たったと判定しています。ボールの直径よりも距離を短めにしているのは、ぎりぎりの隙間を狙ったときに、撃ったボールが隙間を抜けるようにしたかったからです。ゲームを遊びやすくするには、実際にゲームをプレイしながら、距離を調整するとよいでしょう。

ボールが接触したら、撃ったボールをぶら下げる位置を決めます(Fig. 5-40)。本書のサンプルでは、ぶら下がったボールに隣接する空のセルのなかで、撃ったボールに最も近いセルに、撃ったボールを固定します。撃ったボールとセルの中心間の距離を計算して、距離が最も短いセルを選びます。

ボールセルは互い違いに並んでいるので、ぶら下がったボールに隣接するセルを調べるときには注意が必要です(Fig. 5-41)。隣接するセルは左・右・左上・右上・左下・右下の6個ですが、ボールが奇数段にあるときと、偶数段にあるときでは、セルの相対座標が変わります。

Fig. 5-39 撃ったボールとぶら下がったボールの当たり判定

ぶら下がったボール
撃ったボール
ボールの中心間の距離が
一定値よりも小さいときに
(例えばボールの直径の
80%未満のときに)
当たったと判定する

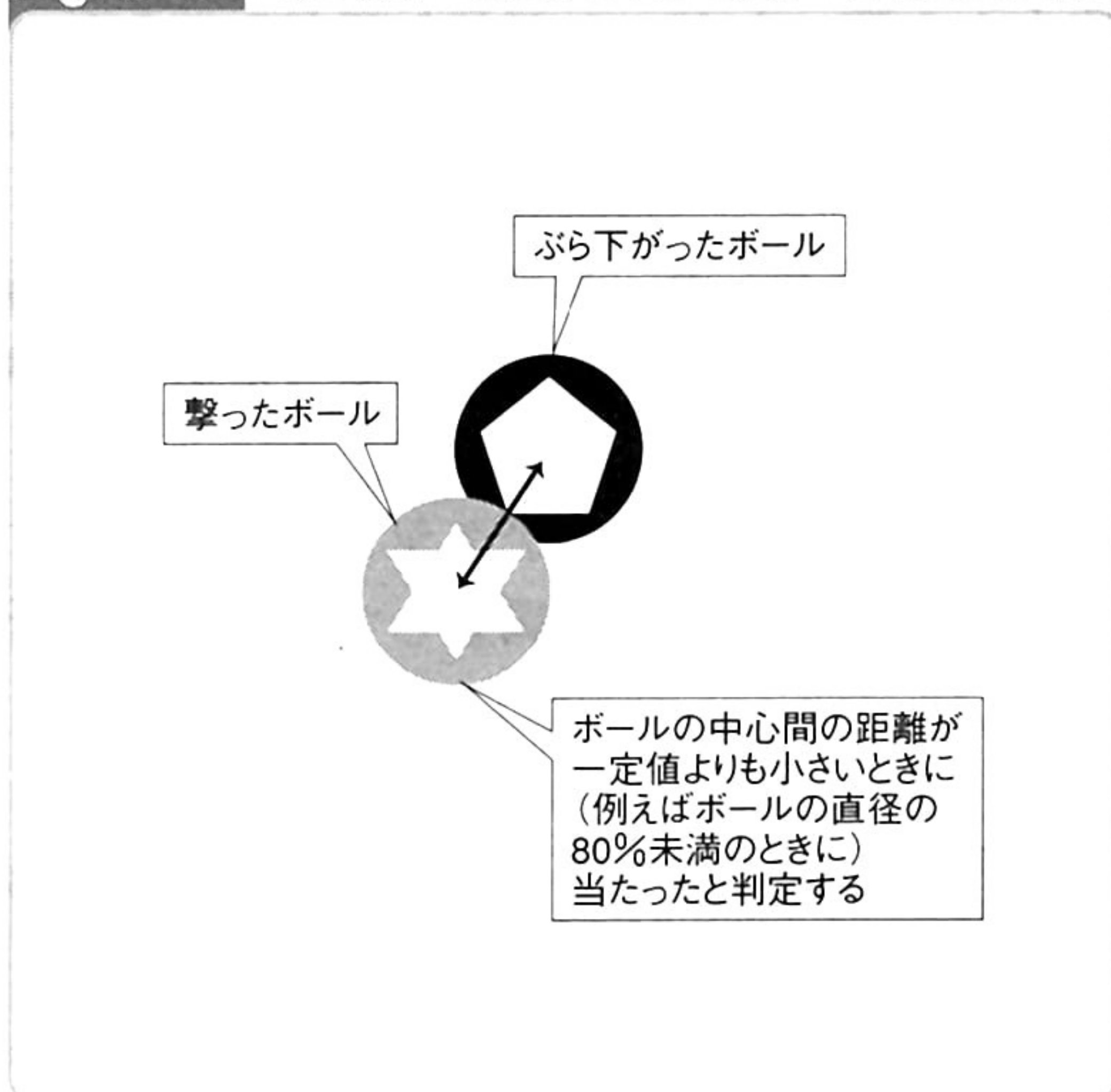
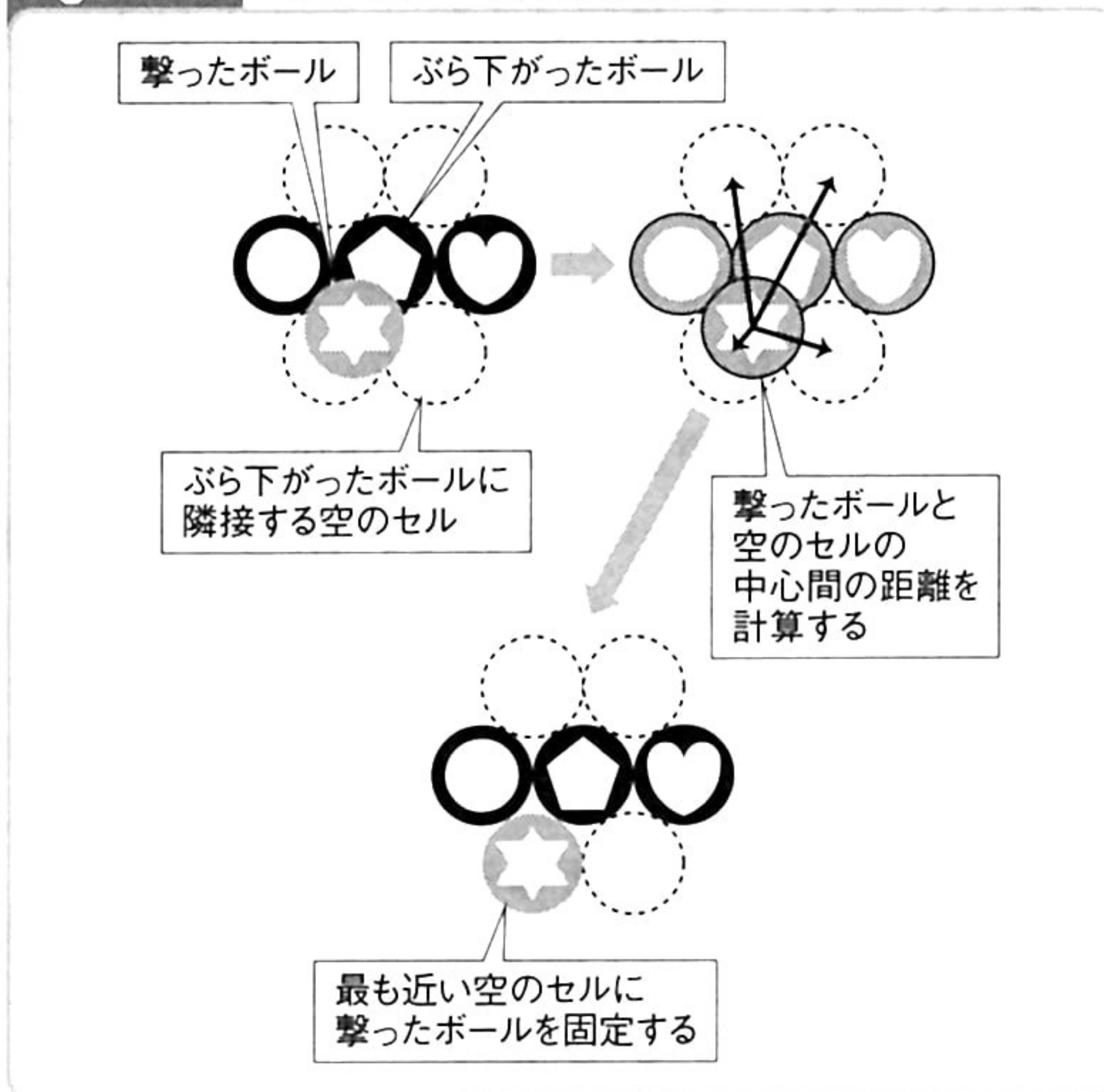
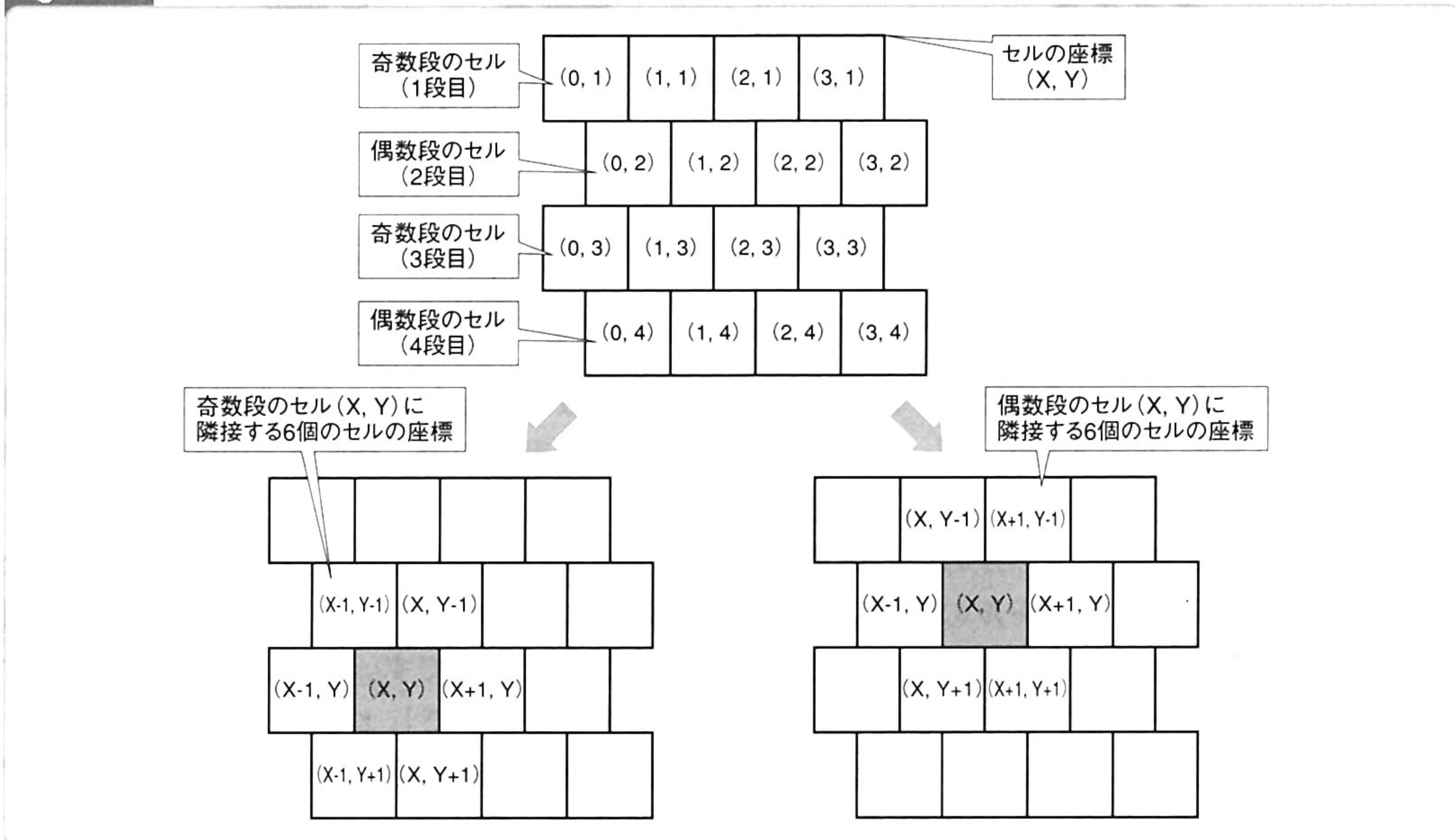
Fig. 5-40 撃ったボールを固定する

撃ったボール
ぶら下がったボール
ぶら下がったボールに
隣接する空のセル
撃ったボールと
空のセルの
中心間の距離を
計算する
最も近い空のセルに
撃ったボールを固定する

Fig. 5-41 隣接するセル

奇数段のセル(1段目): (0, 1) (1, 1) (2, 1) (3, 1)
偶数段のセル(2段目): (0, 2) (1, 2) (2, 2) (3, 2)
奇数段のセル(3段目): (0, 3) (1, 3) (2, 3) (3, 3)
偶数段のセル(4段目): (0, 4) (1, 4) (2, 4) (3, 4)
セルの座標(X, Y)

奇数段のセル(X, Y)に
隣接する6個のセルの座標
(X-1, Y-1) (X, Y-1)
(X-1, Y) (X, Y) (X+1, Y)
(X-1, Y+1) (X, Y+1)

偶数段のセル(X, Y)に
隣接する6個のセルの座標
(X, Y-1) (X+1, Y-1)
(X-1, Y) (X, Y) (X+1, Y)
(X, Y+1) (X+1, Y+1)

プログラム

List 5-8は撃ったボールをぶら下げるプログラムです。ステージの移動処理と、隣接するセルの座標を求める処理を掲載しました。

移動処理（Move関数）では、撃ったボールがぶら下がったボールに接触したかどうかを調べます。ボールの中心間の距離が、一定値（ボールの直径の80%）未満ならば、接触したと判定します。

ボールが接触したら、ぶら下がったボールの周囲の空のセルのなかから、撃ったボールに最も近いセルを選びます。セルとボールの中心間の距離を求めて、最も距離が短いセルに、撃ったボールを固定します。

隣接するセルの座標を求める処理は、X座標の処理（GetAdjacentCX関数）とY座標の処理（GetAdjacentCY関数）に分かれています。どちらの関数も、指定したセルに隣接する6個のセルのうち、指定したセルの座標を返します。

X座標については、セルが奇数段か偶数段かによって、隣接するセルの相対座標が変わります。このプログラムでは、奇数段用と偶数段用の座標を、別々のテーブルにしています。

List 5-8 撃ったボールがぶら下がる（CHangingBallStageクラス）

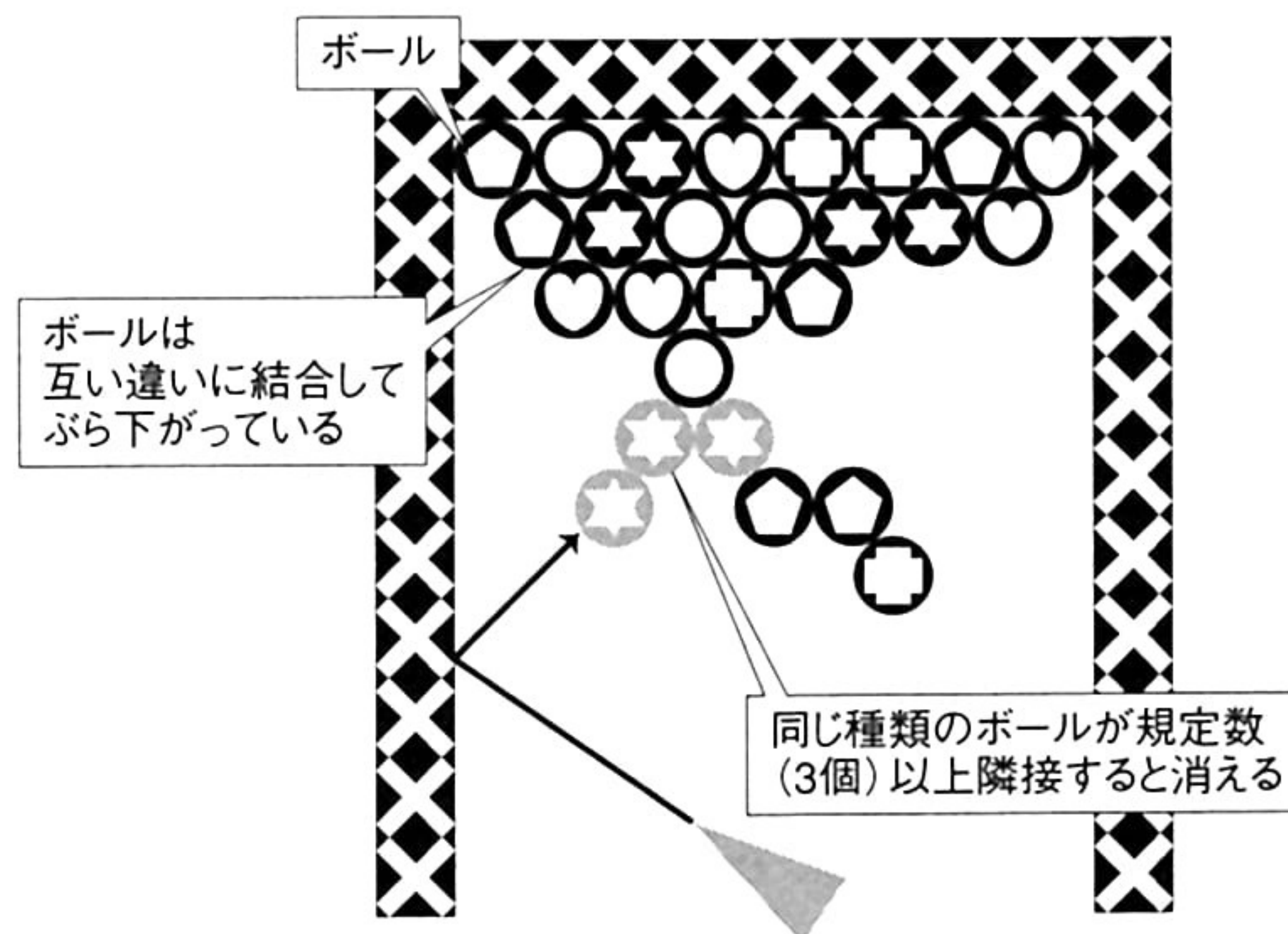
```
// 移動処理
bool CHangingBallStage::Move(const CInputState* is) {

    // ...（中略）...

    // 射出状態
    if (State==2) {

        // ...（中略）...

        // 撃ったボールがぶら下がったボールに
        // 接触したかどうかを調べる
        int x, y;
        for (y=0; y<bys; y++) {
            for (x=0; x<bxs; x++) {

                // 撃ったボールとぶら下がったボールの
                // 中心間の距離を求める
                float
                    dx=BX-GetBallX(x, y),
                    dy=BY-GetBallY(y);

                // 中心間の距離が
                // ボールの直径の80%未満ならば、
                // 接触したとみなす
                if (
```

```
                IsBall(BallCell->Get(x, y)) &&
                dx*dx+dy*dy<0.8f
            ) break;
        }
        if (x<bxs) break;
    }

    // 撃ったボールがぶら下がったボールに接触したら、
    // 撃ったボールをぶら下げる
    if (y<bys) {

        // ぶら下がったボールに隣接する空のセルのなかから、
        // 撃ったボールに最も近いセルを選ぶ
        float d=1000;
        int nx=-1, ny=-1;
        for (int i=0; i<6; i++) {

            // 隣接するセルの座標を取得する
            int
                cx=GetAdjacentCX(x, y, i),
                cy=GetAdjacentCY(x, y, i);

            // セルと撃ったボールの
            // 中心間の距離を計算する
            float
                ex=BX-GetBallX(cx, cy),
                ey=BY-GetBallY(cy),
                e=sqrtf(ex*ex+ey*ey);

            // セルが空で、
            // 距離が他のセルよりも短ければ、
            // セルの座標を記録する
            if (
                BallCell->Get(cx, cy)==' ' &&
                d>e
            ) {
                d=e;
                nx=cx;
                ny=cy;
            }
        }

        // 最も近いセルが見つかったら、
        // 撃ったボールをそのセルに固定する
        if (nx>=0) {

            // ボールを固定する
            BallCell->Set(nx, ny, '0'+BType);

            // 初期状態に移行する
```

```
                State=0;

                //同じ種類のボールが並んだかどうかを調べる
                TryEraseBall();
            }
        }
    }

    //...(中略)...
}

//隣接するセルのX座標を求める処理
int CHangingBallStage::GetAdjacentCX(int cx, int cy, int index) {

    //セルが偶数段・奇数段のときの、
    //隣接するセルのX座標のテーブル
    static const int ax[2][6]={
        {-1, 1, 0, 1, 0, 1},
        {-1, 1, -1, 0, -1, 0}
    };

    //隣接するセルのX座標を1つ返す
    return cx+ax[cy%2][index];
}

//隣接するセルのY座標を求める処理
int CHangingBallStage::GetAdjacentCY(int cx, int cy, int index) {

    //隣接するセルのY座標のテーブル
    //値の順番は、X座標のテーブルに対応している
    static const int ay[]=
        { 0, 0, -1, -1, 1, 1};

    //隣接するセルのY座標を1つ返す
    return cy+ay[index];
}
```

ぶら下がった同じ種類のボールを消す

「ぶら下がったボール」(→p. 301) において、同じ種類のボールが規定数 (3個) 以上隣接したときに、ボールを消すアクションです。「撃ったボールがぶら下がる」(→p. 311) で解説したように、適切な位置にボールを撃ち込んでぶら下げることによって、ボールの種類を揃えて消すことができます。

Fig. 5-42 同じ種類の隣接したボールが消える

撃ち込んだボールがぶら下がったときに、同じ種類のボールが規定数以上隣接していたら、ボールが消えます (Fig. 5-42)。本書のサンプルでは、ボールが時間とともにだんだん薄くなり、一定時間が経過すると完全に消えます。

ぶら下がった同じ種類のボールを消すアクションは、『パズルボブル』に採用されています。このゲームでは、ボールを消したときに、そのボールにぶら下がっていた他のボールが支えを失って落ちてきます。ブドウの房のようにボールがぶら下がっているときに、根元に近いボールを消すと、多くのボールを一気に落として消すことができます。根元を狙うのは難しいのですが、大量のボールを落とすのは、なんとも爽快です。

アルゴリズム

ぶら下がった同じ種類のボールを消すには、すべてのぶら下がったボールについて、隣接する同じ種類のボールの数を調べます (Fig. 5-43)。同じ種類のボールが隣接していたら、さらに

Fig. 5-43 隣接する同じ種類のボールを数える

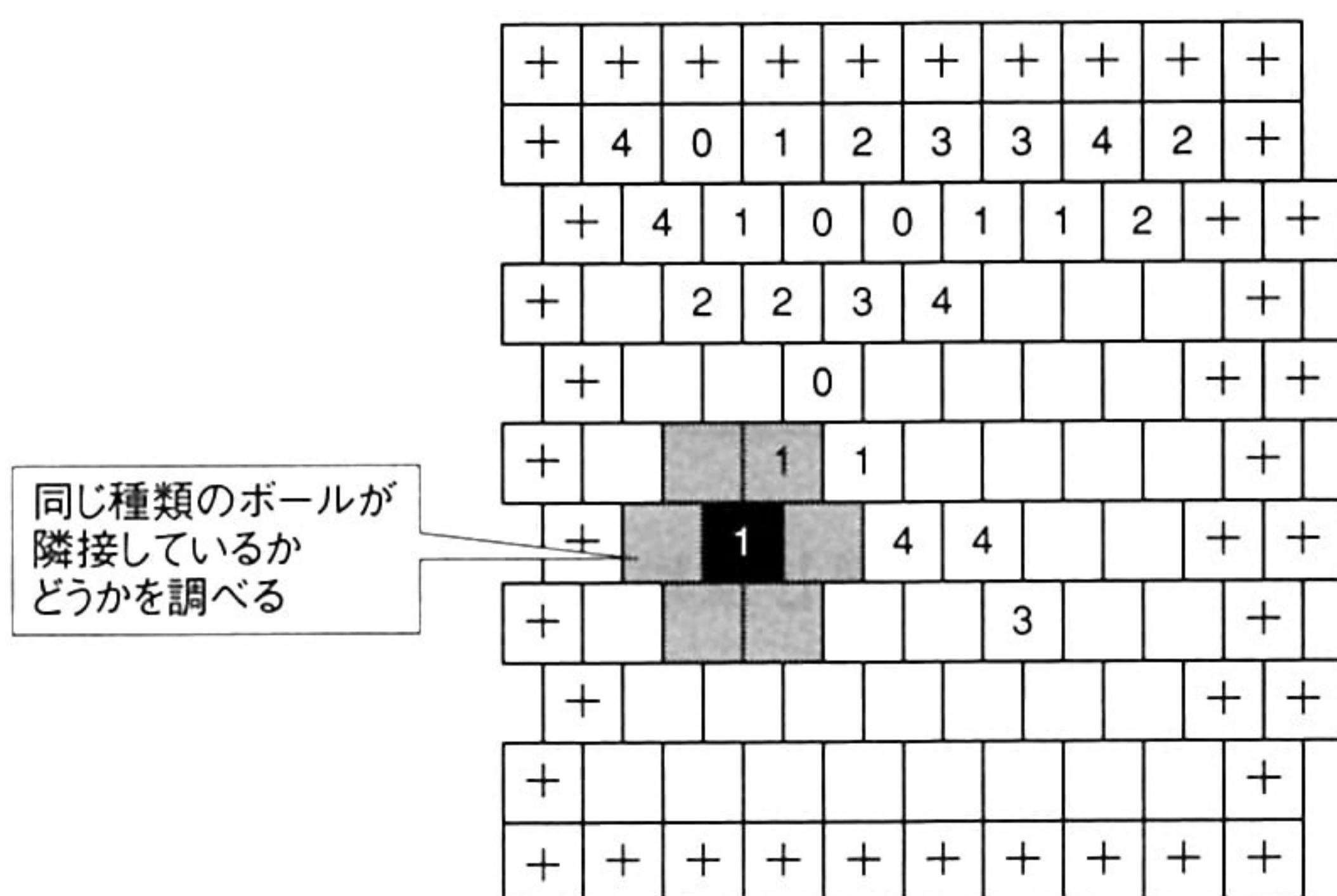
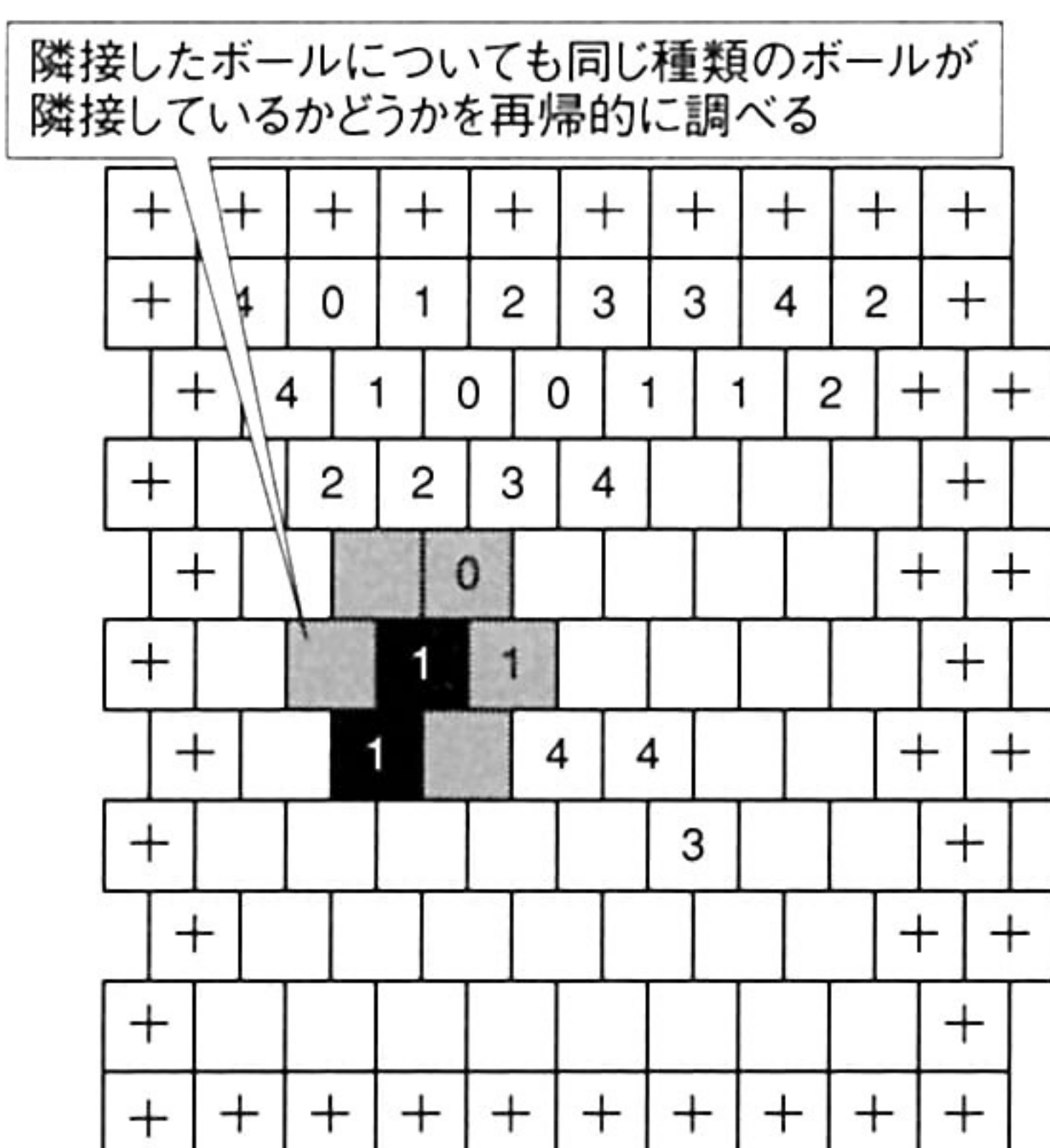
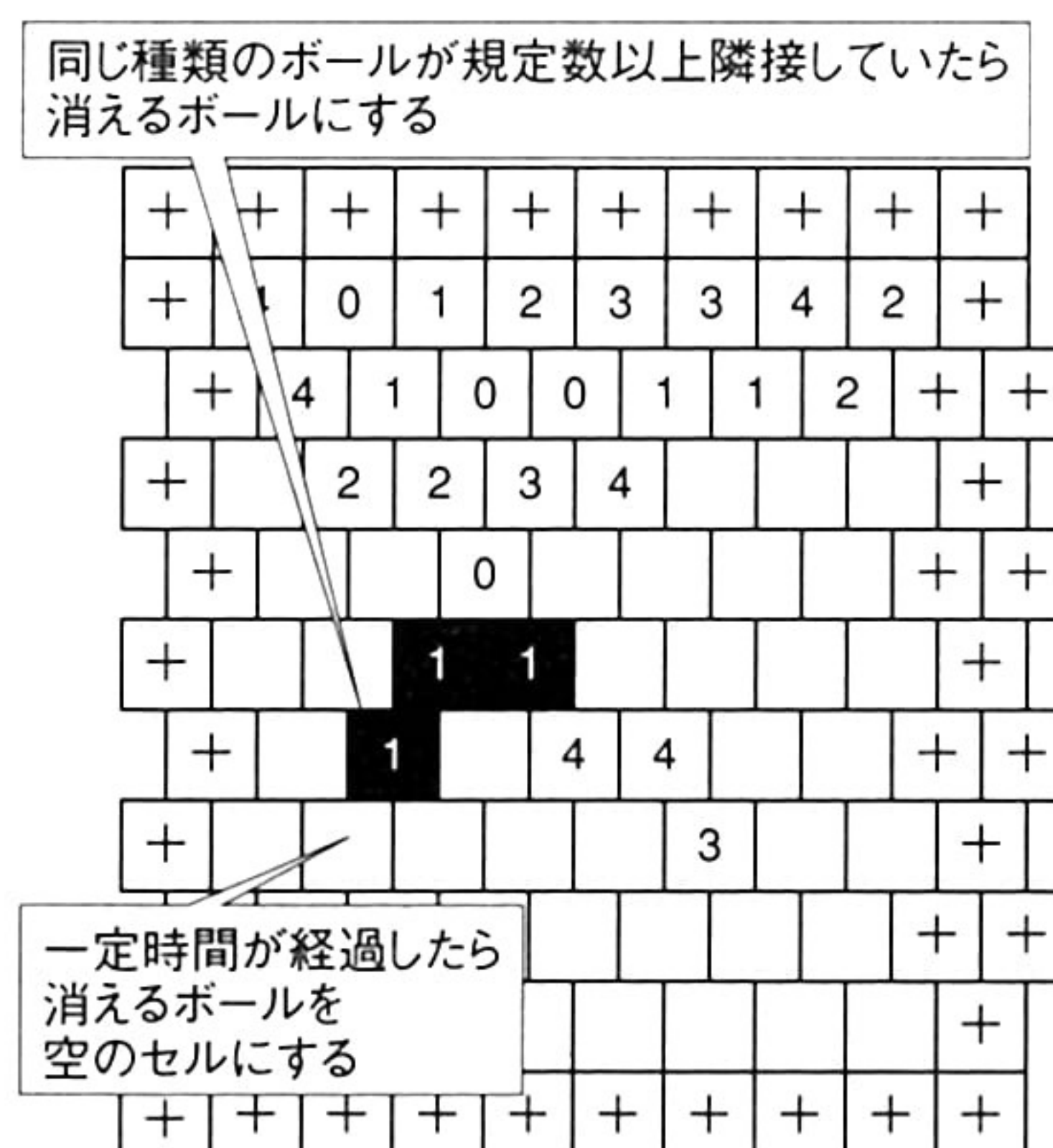



Fig. 5-44　再帰的に同じ種類のボールを数える

隣接したボールについても同じ種類のボールが隣接しているかどうかを再帰的に調べる

Fig. 5-45　ボールを消す

同じ種類のボールが規定数以上隣接していたら消えるボールにする

一定時間が経過したら消えるボールを空のセルにする

そのボールに隣接するボールについても、再帰的に数えます（Fig. 5-44）。

同じ種類のボールが規定数（3個）以上隣接していたら、それらのボールを消します（Fig. 5-45）。本書のサンプルでは、ボールを消えるボールとしてマークします。消えるボールは時間とともに表示を薄くして、一定時間が経過したら、空のセルに変化させます。

プログラム

List 5-9はぶら下がった同じ種類のボールを消すプログラムです。消えるボールを探す処理、ボールを消す処理、ステージの移動処理を掲載しました。

消えるボールを探す処理（TryEraseBall関数）では、ステージ内のすべてのボールについて、同じ種類のボールが規定数（3個）以上隣接しているかどうかを調べます。隣接していたら、それらのボールを消えるボールに変化させます。

隣接しているボールを数える処理は、ボールを消す処理（EraseBall関数）で行います。この関数は、ボールを数える処理と、ボールを消えるボールに変える処理を兼ねています。いずれの場合も、隣接したセルの種類を調べて、同じ種類のボールならば、セルをマークします。

ボールを数えるときには、数えたボールのセルの最上位ビットを「1」にすることで、カウントずみのマークを付けます。消すときには、ボールのセルの上位から2ビット目を「1」にすることで、消えるボールのマークを付けます。

隣接したセルに隣接するセルに関しても、同じ種類のボールかどうかを調べます。もしも同じ種類ならば、さらに隣接するセルについても同様の処理を再帰的に行います。このようにして、隣接した同じ種類のボールをすべて探し出します。

移動処理（Move関数）では、一定時間が経過するのを待ってから、消えるボールを空のセルに変えることによって、完全に削除します。一定時間が経過するまでに、タイマーを使ってボールの表示をだんだん薄くすることによって、ボールが滑らかに消える演出を行います。ボー

ルの描画については、「ぶら下がったボール」(→p. 301) の描画処理 (Draw関数) を参照してください。

List 5-9　ぶら下がった同じ種類のボールを消す(CHangingBallStageクラス)

```
//消えるボールを探す処理
void CHangingBallStage::TryEraseBall() {

    //すべてのボールについて、
    //同じ種類のボールが規定数(3個)以上
    //隣接しているかどうかを調べる
    int bxs=BallCell->GetXSize(), bys=BallCell->GetYSize();
    for (int y=0; y<bys; y++) {
        for (int x=0; x<bxs; x++) {

            //同じ種類のボールが規定数以上隣接していたら、
            //消えるボールに変化させる
            char c=BallCell->Get(x, y);
            if (
                IsBall(c) &&
                EraseBall(x, y, c, 0x80)>=HANGING_BALL_ERASE
            ) {
                //消えるボールに変化させる
                EraseBall(x, y, c|0x80, 0x40);

                //タイマーを設定し、
                //消去状態に移行する
                Time=0;
                State=3;
            }
        }
    }

    //ボールのカウントずみのマークを解除する
    for (int y=0; y<bys; y++) {
        for (int x=0; x<bxs; x++) {
            BallCell->Set(x, y, BallCell->Get(x, y)&0x7f);
        }
    }
}

//ボールを消す処理
//隣接した同じ種類のボールを数える処理も兼ねている
int CHangingBallStage::EraseBall(int x, int y, char c, int mask) {

    //隣接した同じ種類のセルの数
    int count=0;

    //現在位置のセルが指定された種類のセルならば、
    //セルをマークする
```

```
	if (BallCell->Get(x, y)==c) {

		// 指定されたビットをセットすることによって、
		// セルをマークする
		BallCell->Set(x, y, c|mask);

		// 現在位置のセルを、
		// 隣接した同じ種類のセルの数に加える
		count++;

		// 周囲のセルについても再帰的に調べて、
		// 隣接した同じ種類のセルの数を集計する
		for (int i=0; i<6; i++) {
			count+=EraseBall(
				GetAdjacentCX(x, y, i),
				GetAdjacentCY(x, y, i),
				c, mask);
		}
	}

	// 同じ種類のセルの数を返す
	return count;
}

// 移動処理
bool CHangingBallStage::Move(const CInputState* is) {

	// ...(中略)...

	// 消去状態
	if (State==3) {

		// 一定時間が経過したら、
		// 消えるボールを完全に削除する
		Time++;
		if (Time==30) {
			for (int y=0; y<bys; y++) {
				for (int x=0; x<bxs; x++) {

					// 消えるボールを空のセルにする
					if (BallCell->Get(x, y)&0x40) {
						BallCell->Set(x, y, ' ');
					}
				}
			}

			// 初期状態に移行する
			State=0;
		}
	}
```

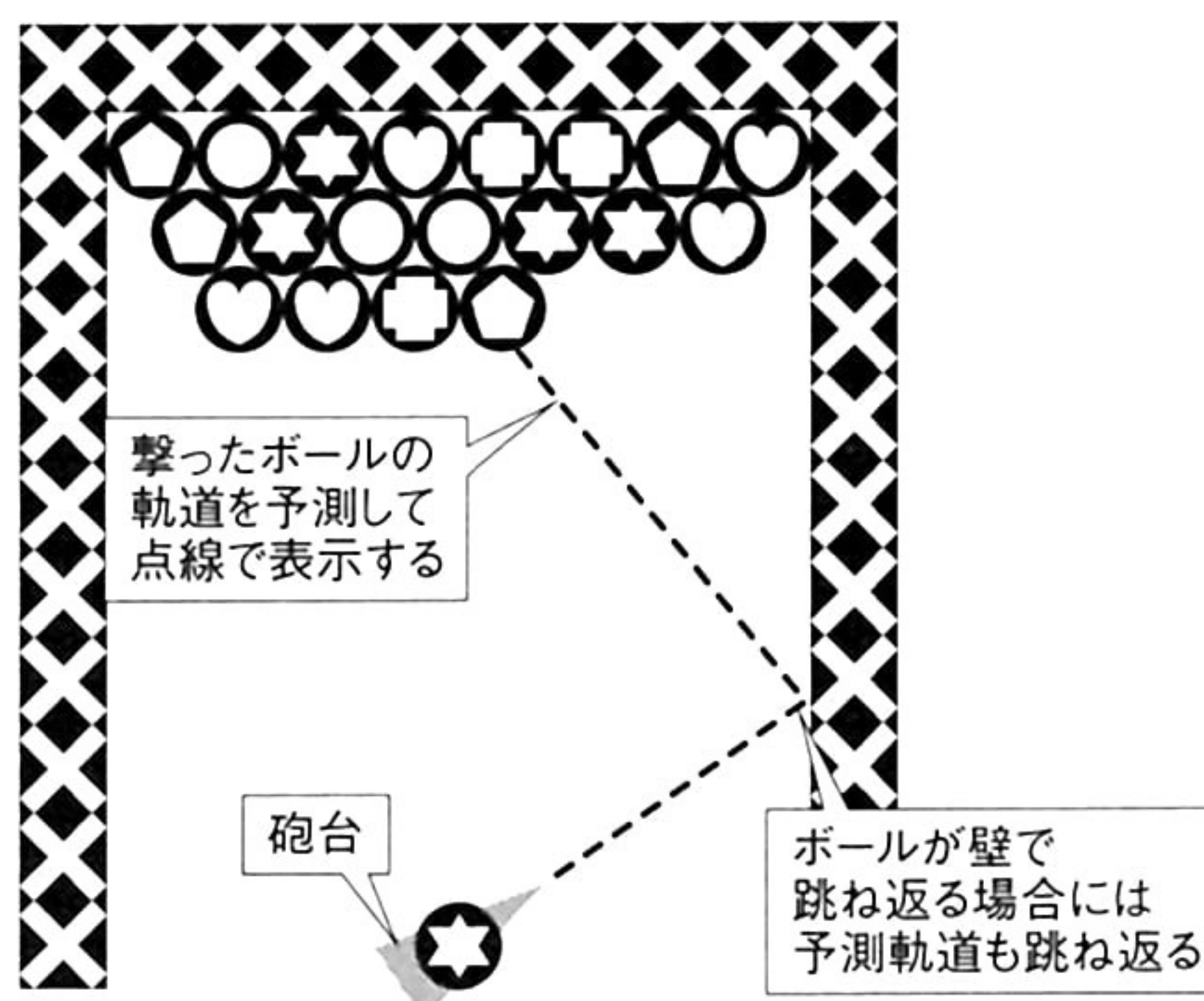
```
        return true;
    }
```

ボールの軌道を予測して表示する

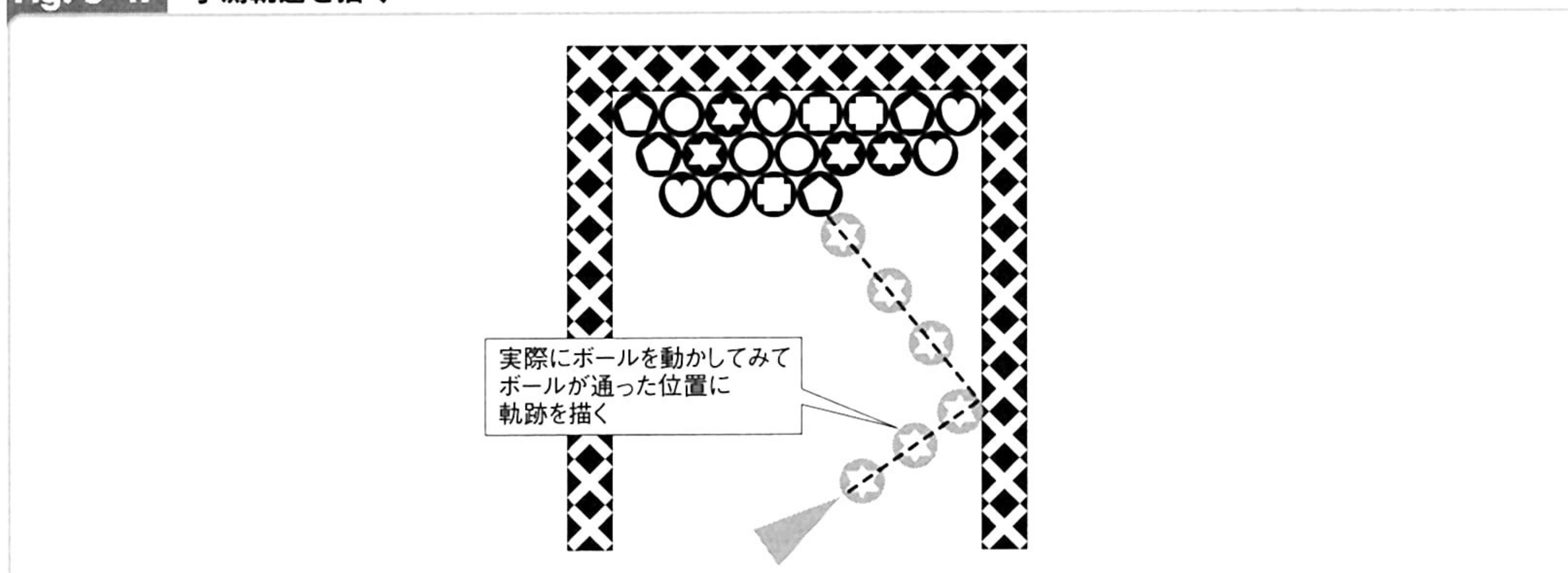
「撃ったボールが跳ね返る」(→p. 306) において、ボールを撃つ前に、ボールの軌道を予測して表示する機能です。ボールが壁で跳ね返る様子もわかるので、目標を正確に狙うことができます。

本書のサンプルでは、ボールの予測軌道を点線で表示します (Fig. 5-46)。ボールが壁で跳ね返る場合には、予測軌道も壁で跳ね返ります。砲台を回転させたときには、すぐに新しい予測軌道を表示します。

予測軌道を表示すると、目的の位置を狙うのが簡単になります。実際のゲームでは、常に予測軌道を表示するのではなく、初級のステージだけで表示するとか、アイテムを取ったときだけ表示するといったルールにしてもよいでしょう。

Fig. 5-46 ボールの予測軌道

アルゴリズム

ボールの軌道を予測して表示するには、実際にボールを動かしてみて、ボールが通った位置に軌跡を描きます (Fig. 5-47)。ボールを動かす方法は、「撃ったボールが跳ね返る」(→p. 306) でボールを動かす方法とまったく同じです。ボールを動かす処理をまとめておいて、実際にボールを動かすときと、軌道を予測するときの、両方から利用できるようにするとよいでしょう。

Fig. 5-47　予測軌道を描く

本書のサンプルでは、撃ったボールがぶら下がったボールに衝突するかどうかまでは、予測していません。実際にはボールに当たる場合にも、ボールを突き抜けるような軌跡を表示します。

軌道を予測するときにも、撃ったボールとぶら下がったボールの当たり判定処理を行えば、撃ったボールがどのボールと接触して、どこにぶら下がるのかが予測できます。このような予測を表示してもよいでしょう。ただし、ゲームが大幅に簡単になるので、ゲームに刺激を加える何か別の要素が必要かもしれません。

プログラム

List 5-10はボールの軌道を予測して表示するプログラムです。ステージの描画処理を掲載しました。

最初にボールの座標と速度を保存しておきます。次に、ボールを一定回数だけ、実際に動かします。このプログラムでは30回動かすことにしました。回数を多くすれば、より遠くまで軌道を予測することができます。ただし、多くの場合は途中でぶら下がっているボールに接触するので、あまり遠くまで予測する意味はありません。

ボールを動かしながら、ボールの座標に小さな円を描きます。これで、ボールの軌跡を点線で表示することができます。軌跡を描いたら、最初に保存していた値を使って、ボールの座標と速度を元に戻します。

List 5-10　ボールの軌道を予測して表示する（CHangingBallStageクラス）

```
// 描画処理
void CHangingBallStage::Draw() {

	// ...（中略）...

	// ボールの軌道を予測して表示する
```

```
    if (State==1) {

        // ボールの座標と速度を保存する
        float bx=BX, by=BY, bvx=BVX, bvy=BVY;

        // ボールを一定回数動かす
        for (int i=0; i<30 && State==1; i++) {

            // ボールを動かす
            MoveBall();

            // ボールの座標に小さな円を描くことにより、
            // ボールの軌跡を表示する
            Game->Texture[TEX_ORB]->Draw(
                (BX-0.1f)*sw, (BY-0.1f)*sh, sw*0.2f, sh*0.2f,
                0, 0, 1, 1, COL_LGRAY);
        }

        // ボールの座標と速度を元に戻す
        BX=bx;
        BY=by;
        BVX=bvx;
        BVY=bvy;

        // 状態を元に戻す
        State=1;
    }

    // ...（中略）...
}
```

ボールを拾って集める

キャラクターを動かして、ボールを拾って集めるアクションです。集めたボールを戻し、同じ種類のボールを規定数以上並べると、消すことができます。

ステージ上方にボールが積まれています（Fig. 5-48）。ステージ下方にいるキャラクターは、レバー入力で左右に動きます。

ボタン0（Zキー）を押すと、キャラクターの真上にあるボールを拾うことができます（Fig. 5-49）。拾うのはキャラクターから一番近いボールです。ただし、同じ種類のボールが並んでいるときには、まとめて拾います。

ボールを持っているときには、キャラクターの上方に持っているボールの種類が表示されます。持っているボールと同じ種類のボールの真下で、ボタン0（Zキー）を押すと、さらにボー

Fig. 5-48　ステージの構成

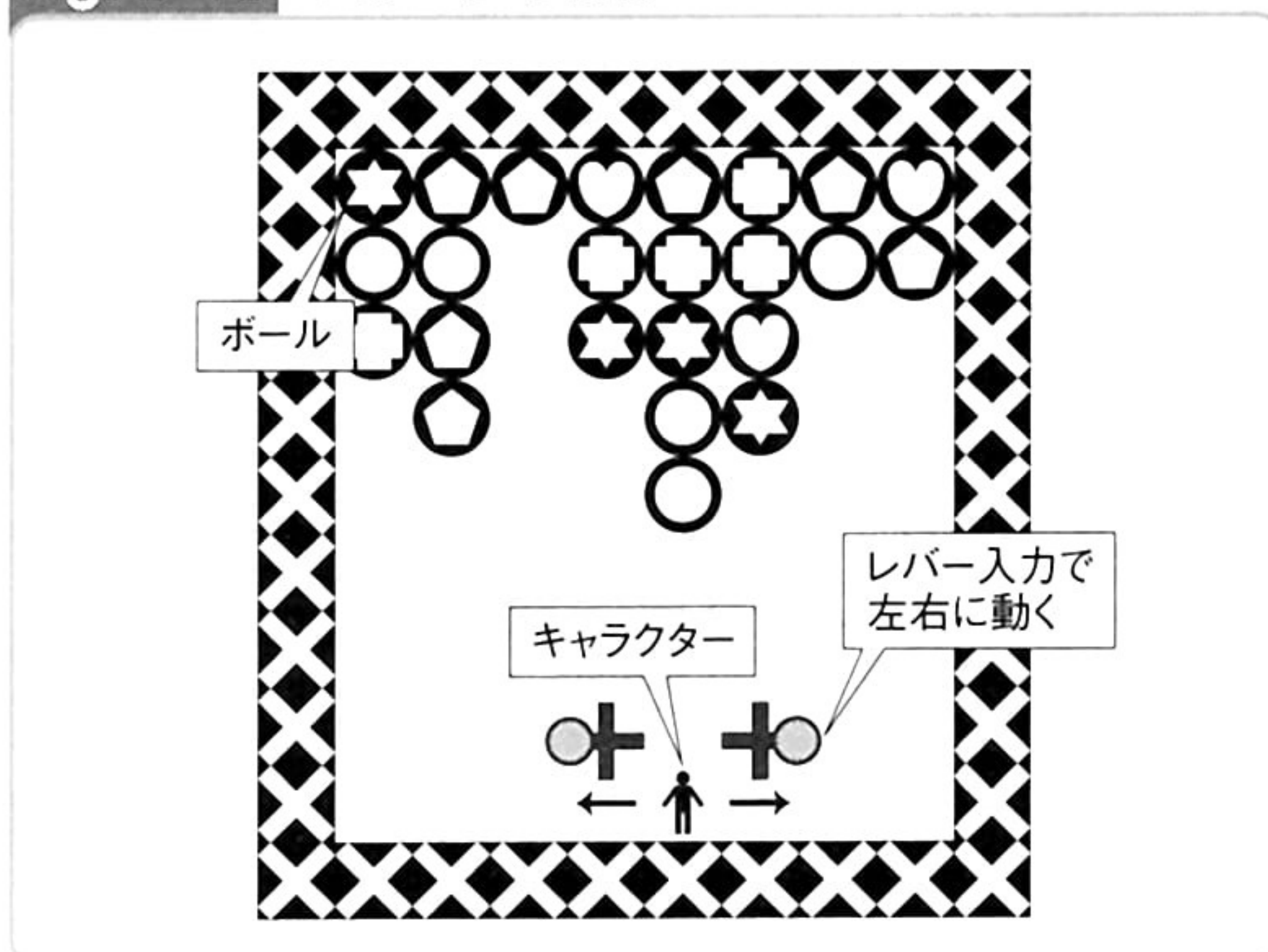



Fig. 5-49　ボールを拾う

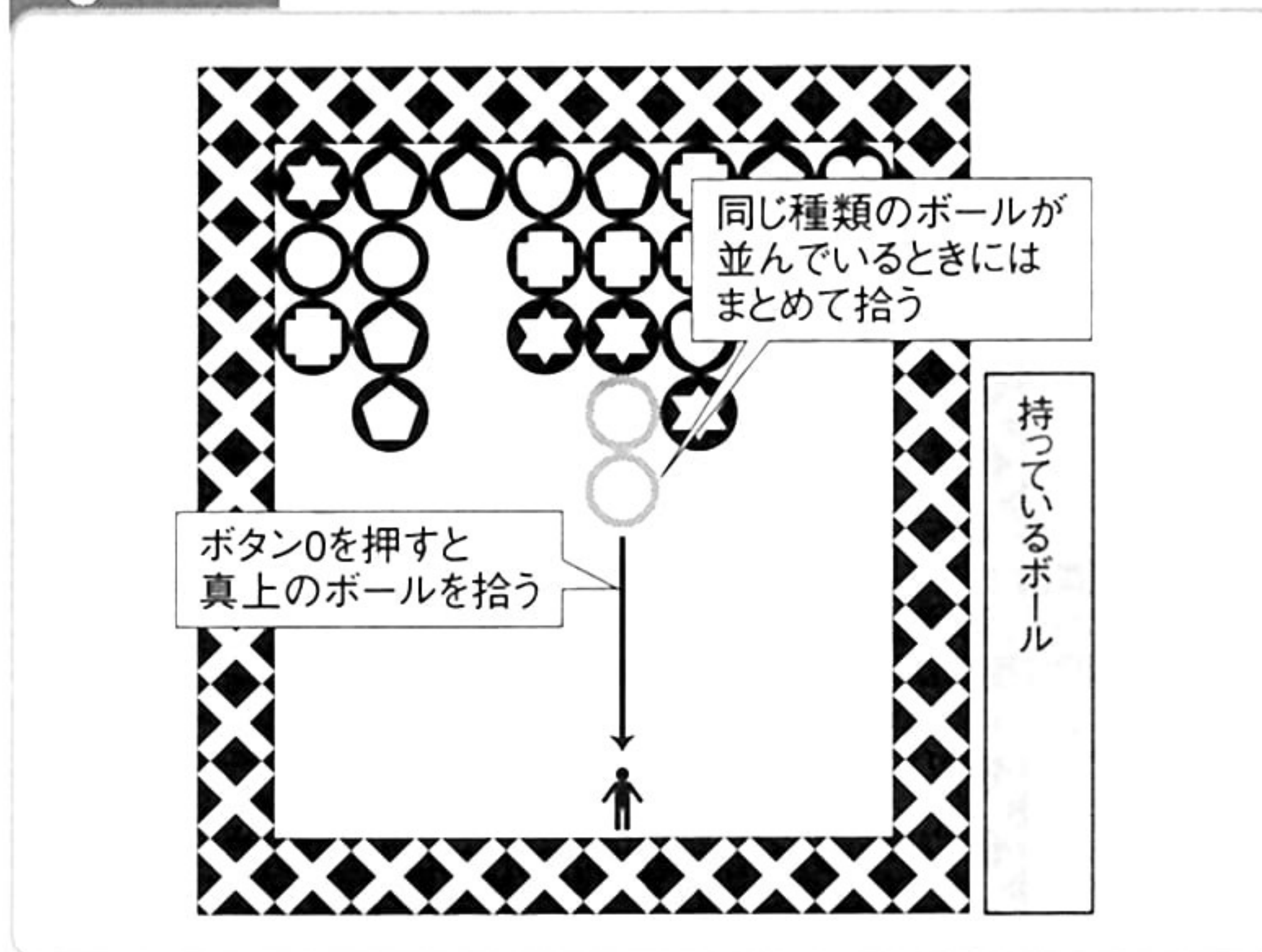



Fig. 5-50　さらにボールを拾う

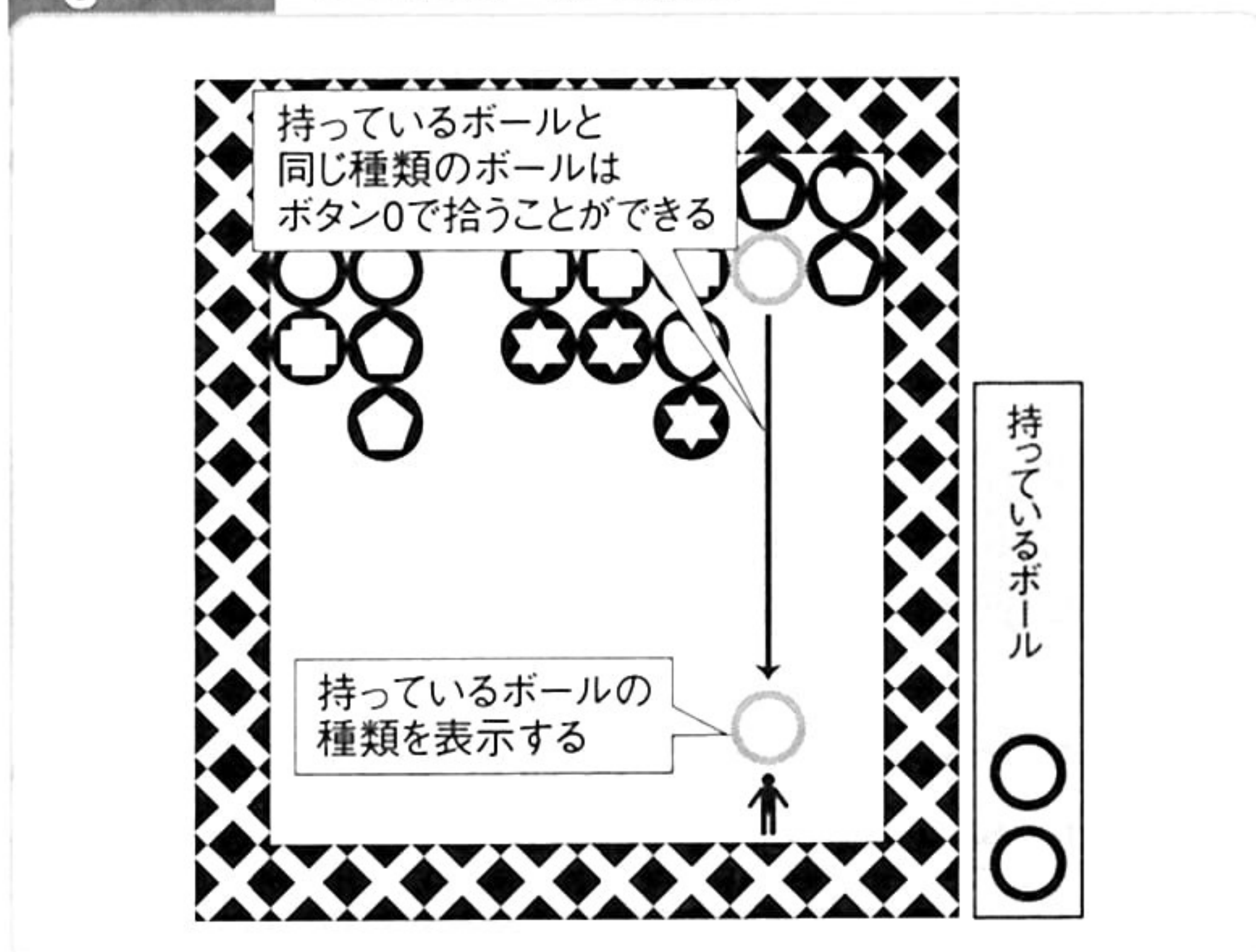
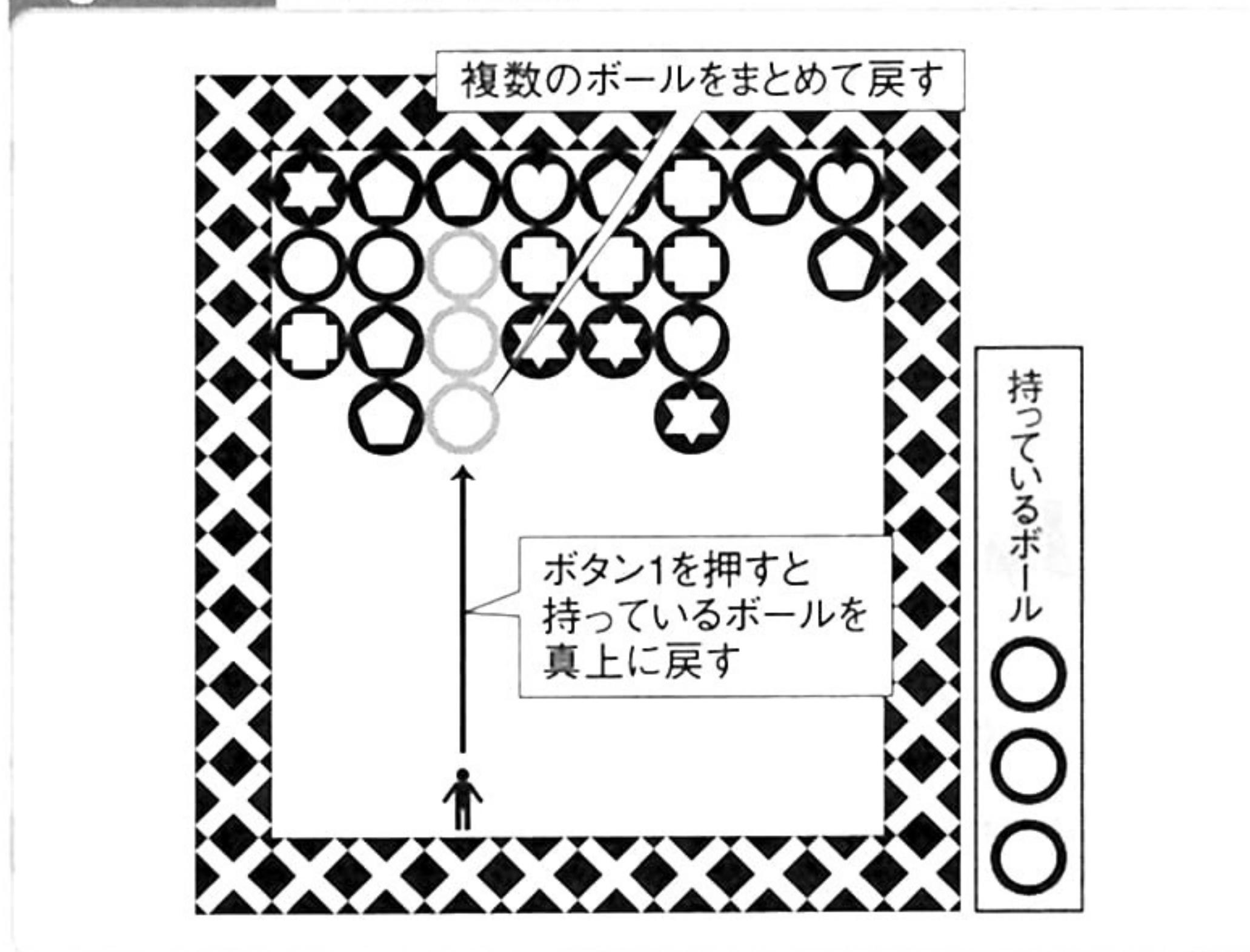



Fig. 5-51　ボールを戻す

ルを拾うことができます (Fig. 5-50)。異なる種類のボールの真下でボタンを押しても、何も起きません。

ボールを持っているときに、ボタン1 (Xキー) を押すと、集めたボールを戻すことができます (Fig. 5-51)。ボールはキャラクターの真上に戻します。複数のボールを持っているときには、複数のボールをまとめて戻します。

ボールを戻したときに、同じ種類のボールが規定数 (3個) 以上縦に並ぶと、ボールを消すことができます (Fig. 5-52)。消えるボールの上下左右に隣接するボールも、一緒に消えます。

ボールを消すと、より下にあったボールが、空いた場所を埋めるように上へ移動します (Fig. 5-53)。このとき、再び同じ種類のボールが規定数以上縦に並ぶと、連鎖的にボールを消すことができます。

ボールを拾って集めるアクションは『マジカルドロップ』に採用されています。このゲームでは、ステージ上方にあるボールが、時間とともに少しずつ落下してきます。ボールがステージ下端に達さないように、次々にボールを消していくことが、ゲームの目的です。

『マネーアイドルエクスチェンジャー』も、ボールを拾って集めるアクションを採用してい

Fig. 5-52 ボールを消す

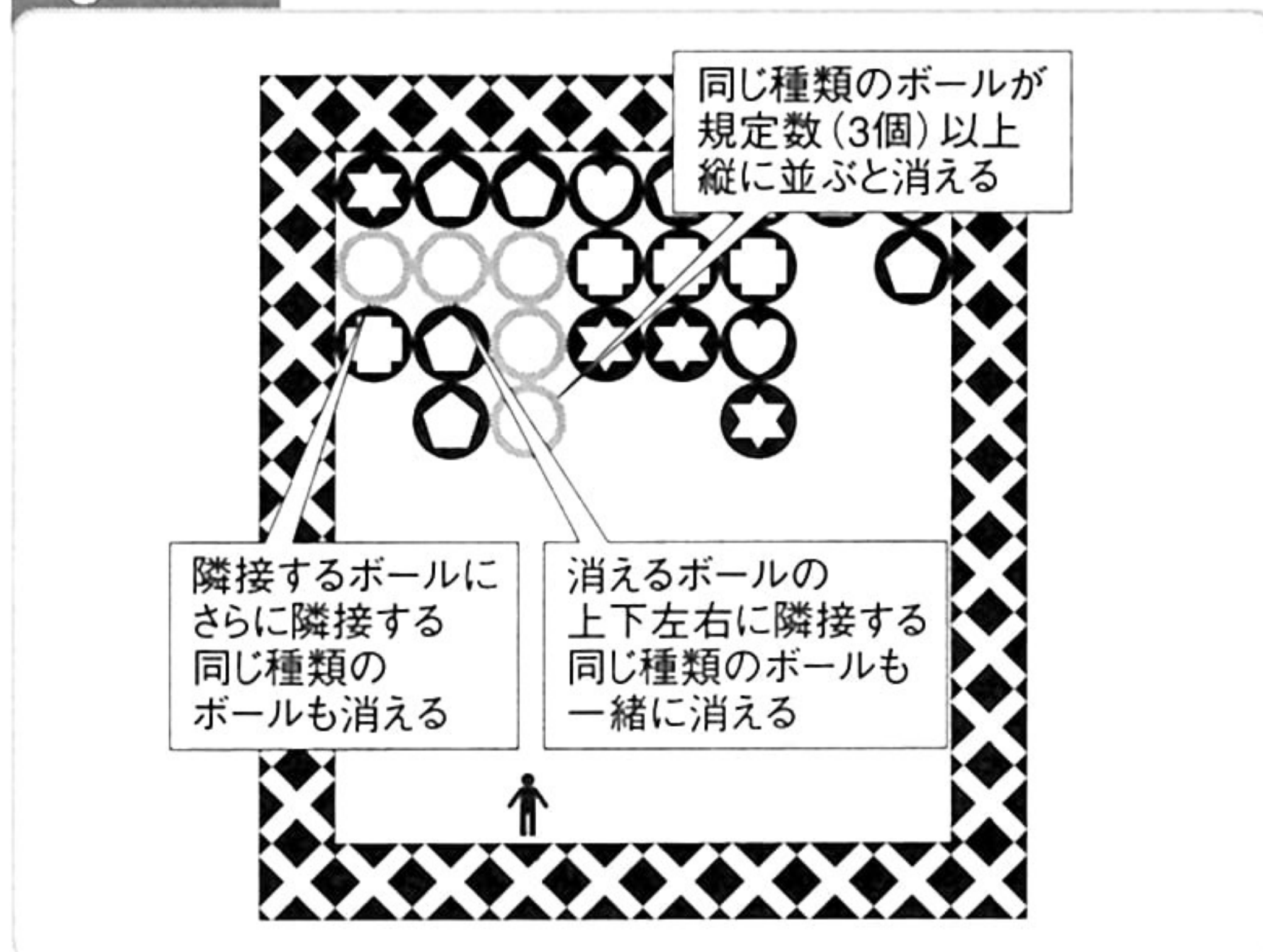
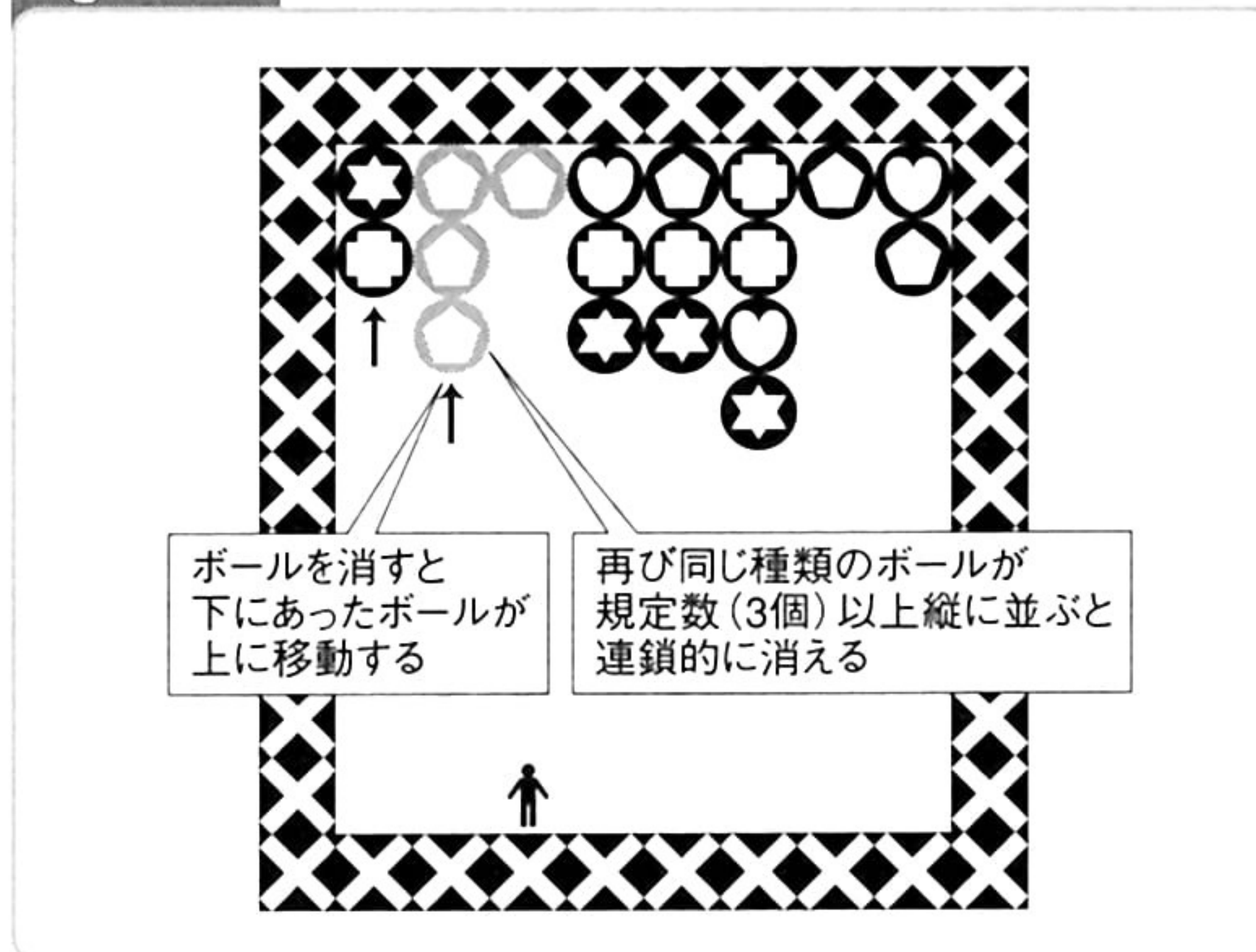
同じ種類のボールが
規定数（3個）以上
縦に並ぶと消える

隣接するボールに
さらに隣接する
同じ種類の
ボールも消える

消えるボールの
上下左右に隣接する
同じ種類のボールも
一緒に消える

Fig. 5-53 連鎖的にボールを消す

ボールを消すと
下にあったボールが
上に移動する

再び同じ種類のボールが
規定数（3個）以上縦に並ぶと
連鎖的に消える

ます。このゲームの特徴は、ボールがコインになっていることです。コインには1円・5円・10円・50円・100円・500円があり、1円・10円・100円は5個以上を、5円・50円は2個以上を隣接させると、上位のコインに変化します。500円を2個以上隣接させると、コインを消すことができます。

アルゴリズム

ボールを拾って集めるアクションを実現するには、ステージをセルで表現します（Fig. 5-54）。ステージの壁は「＝」、ボールは「0〜4」の数字で表しました。

Fig. 5-54 ステージをセルで表現する

壁（＝）

ボール（0〜4）

＝	＝	＝	＝	＝	＝	＝	＝	＝	＝
＝	1	4	4	2	4	3	4	2	＝
＝	0	0		3	3	3	0	4	＝
＝	3	4		1	1	2			＝
＝		4			0	1			＝
＝					0				＝
＝									＝
＝									＝
＝									＝
＝									＝
＝	＝	＝	＝	＝	＝	＝	＝	＝	＝

ボールを拾う

ボタン0（Zキー）を押したら、キャラクターの真上のセルを調べて、一番近いボールを拾います（Fig. 5-55）。ボールを空のセルにして、拾ったボールの種類を記録します。同じ種類のボールが連続しているときには、まとめて拾います。

キャラクターがボールを持っているときには、持っているボールと拾うボールの種類が同じときだけ、ボールを拾うことができます（Fig. 5-56）。種類が異なるときには、ボールを拾いません。ボールを拾ったら、持っているボールの数を増加させます。

Fig. 5-55　一番近いボールを拾う

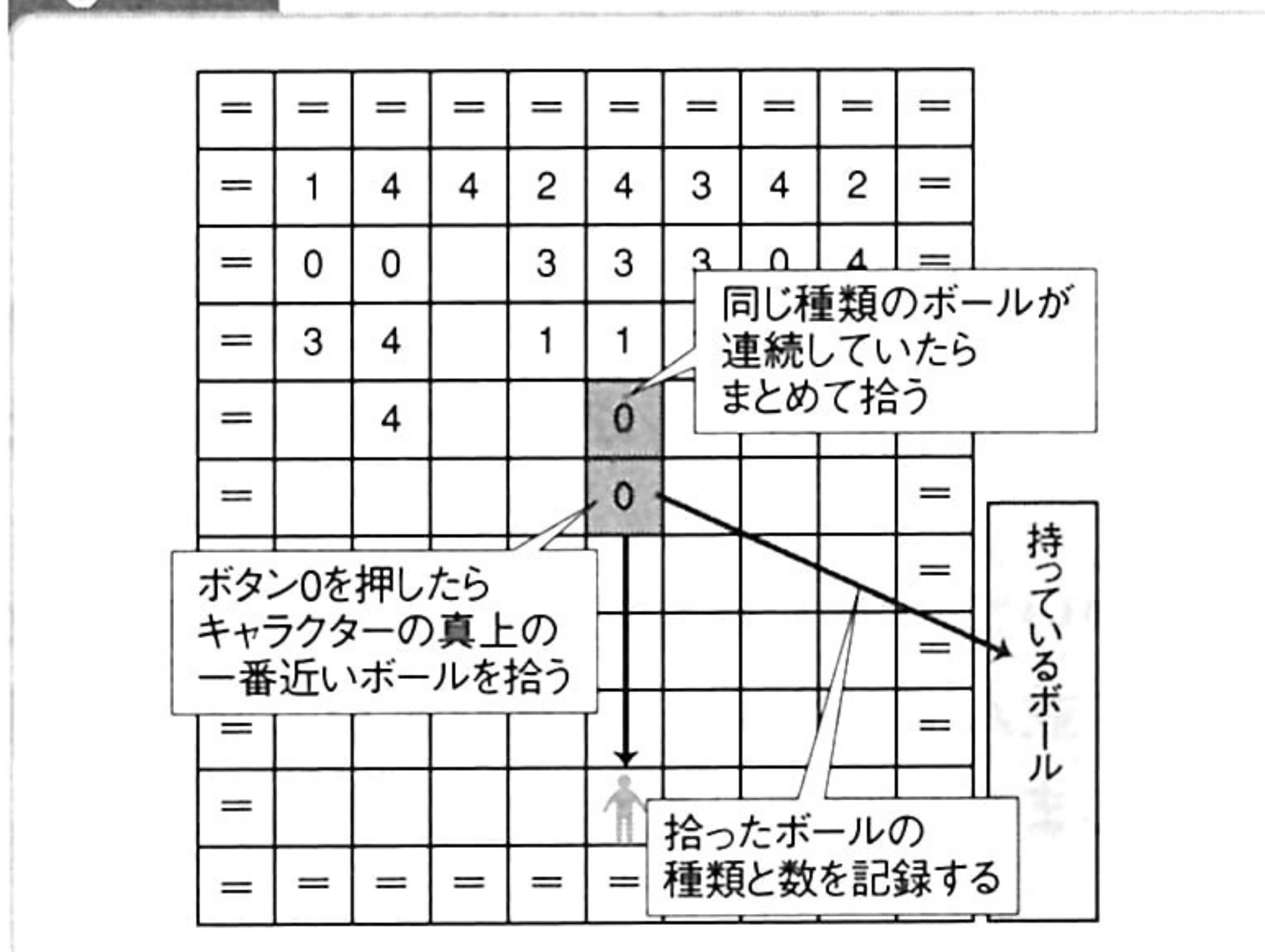
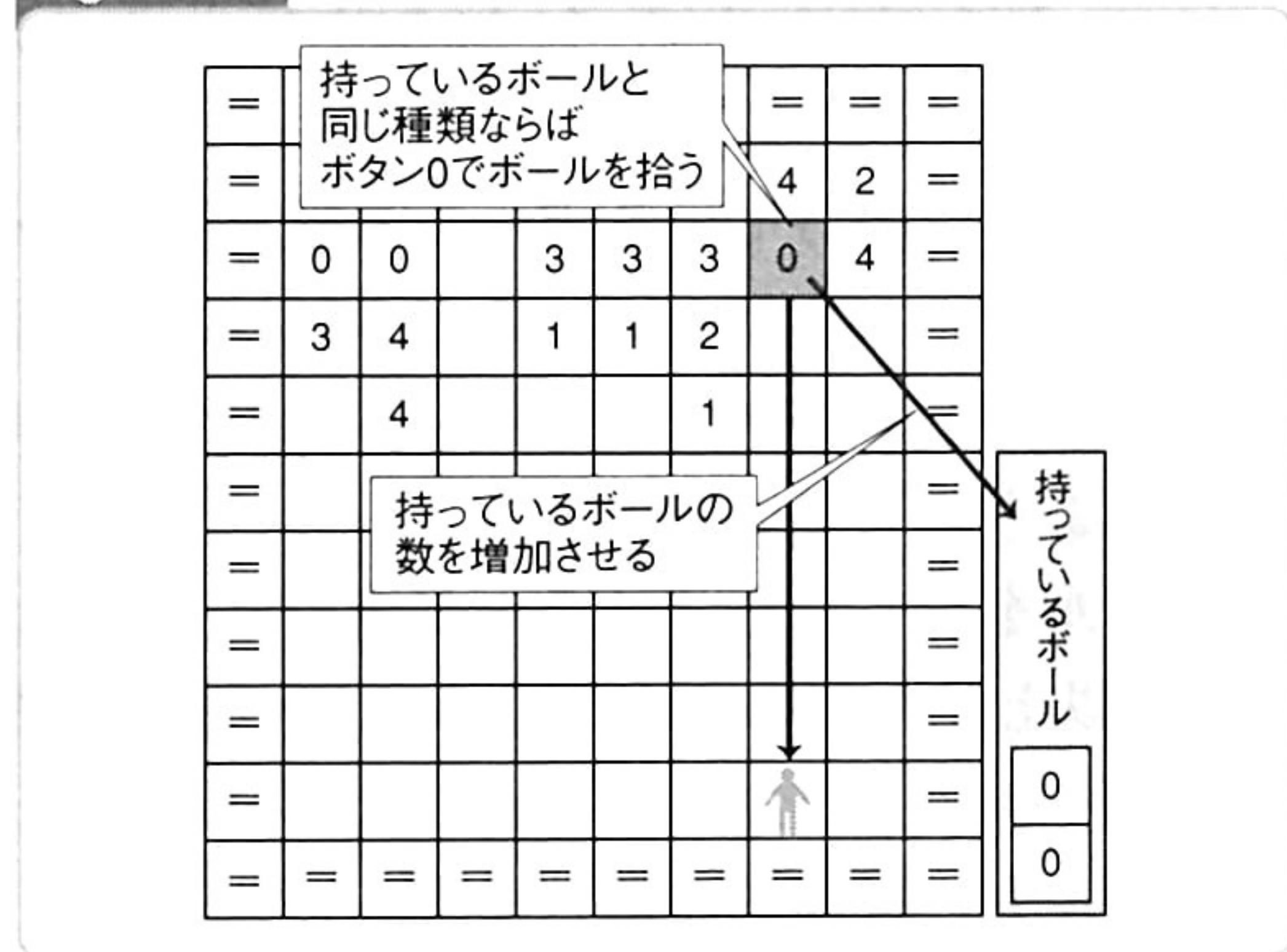



Fig. 5-56　持っているボールと同じ種類のボールを拾う

ボールを戻す

ボタン1（Xキー）を押したら、キャラクターの真上のセルを調べて、一番近いボールまたは壁を探します（Fig. 5-57）。そして、持っているボールの種類を見つかったボールまたは壁の下

Fig. 5-57　一番近いボールの下にボールを戻す

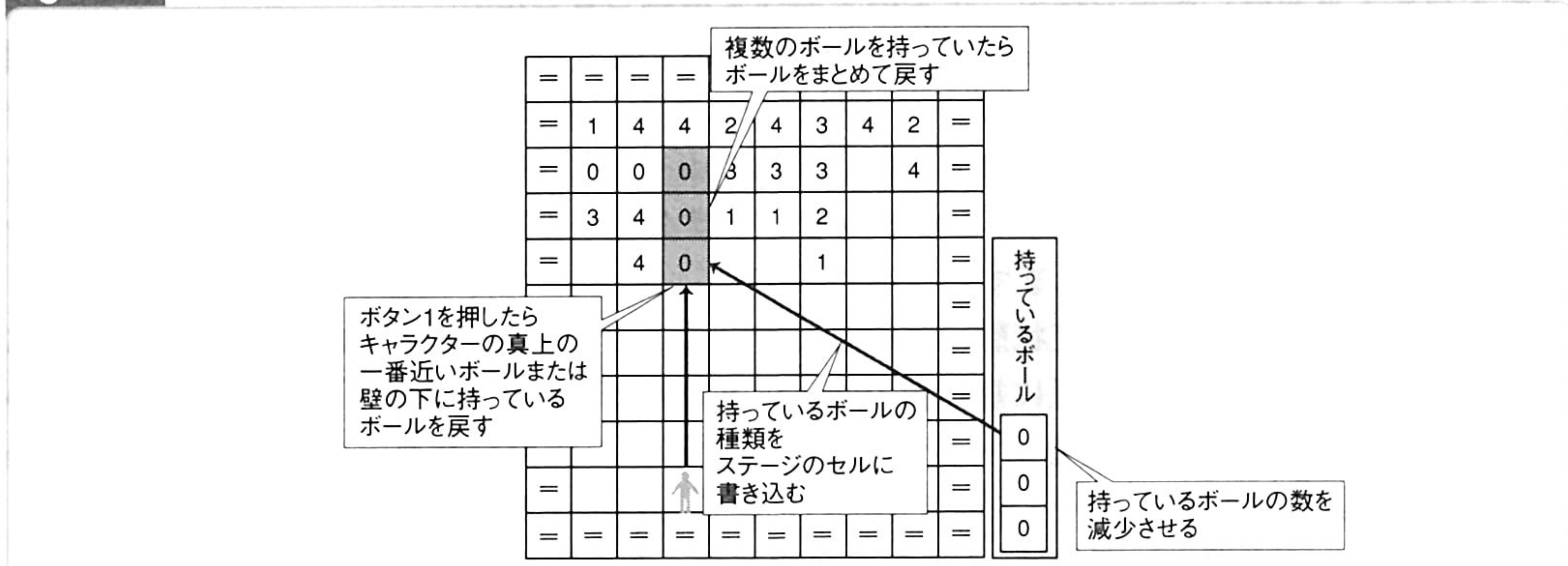



Fig. 5-58 同じ種類のボールが規定数以上並んでいるか調べる

隣接したボールに
隣接した同じ種類の
ボールも再帰的に消す

同じ種類のボールが
縦に規定数（3個）以上
並んでいたら消す

消えるボールの
上下左右に隣接した
同じ種類のボールも消す

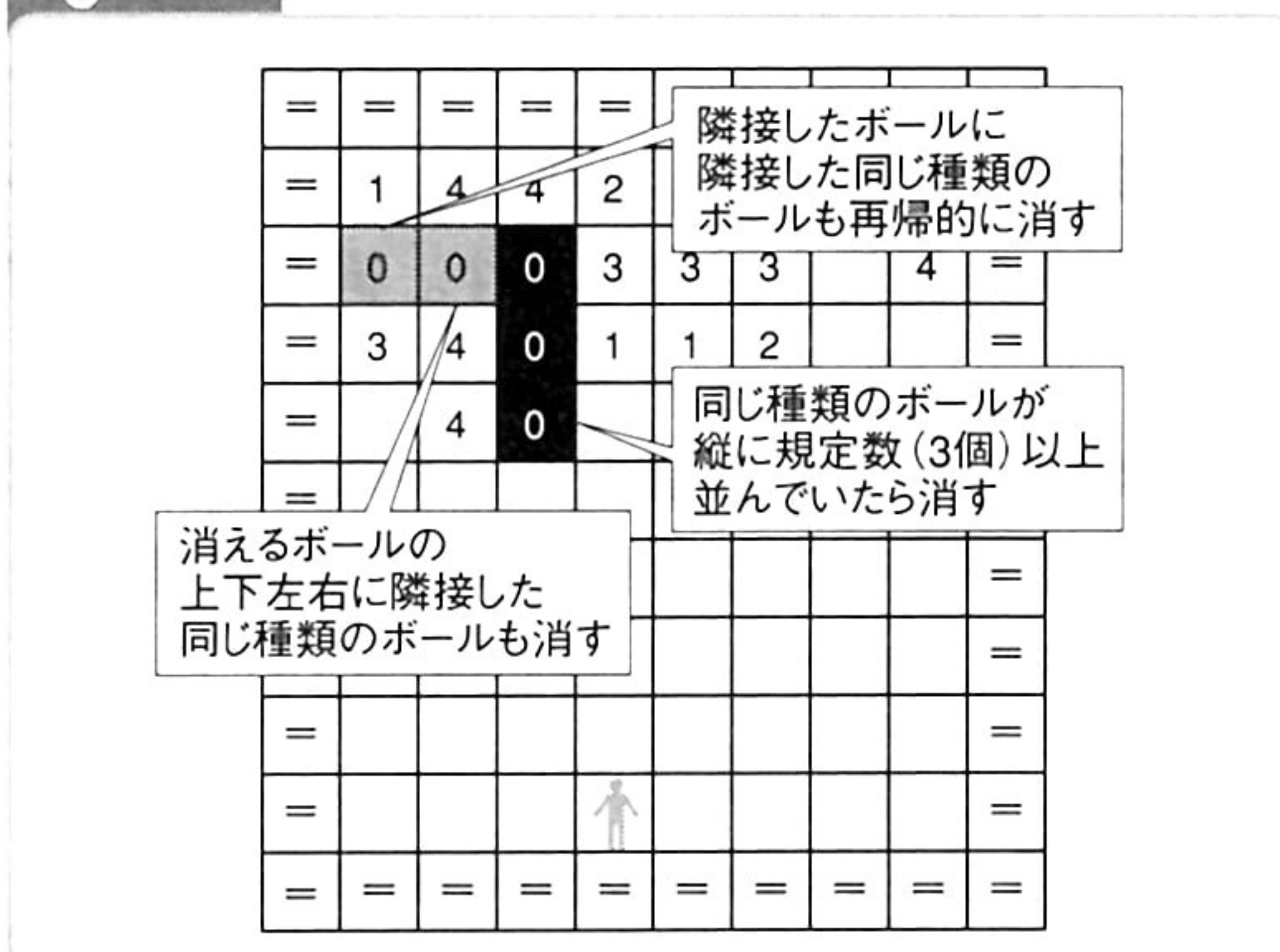
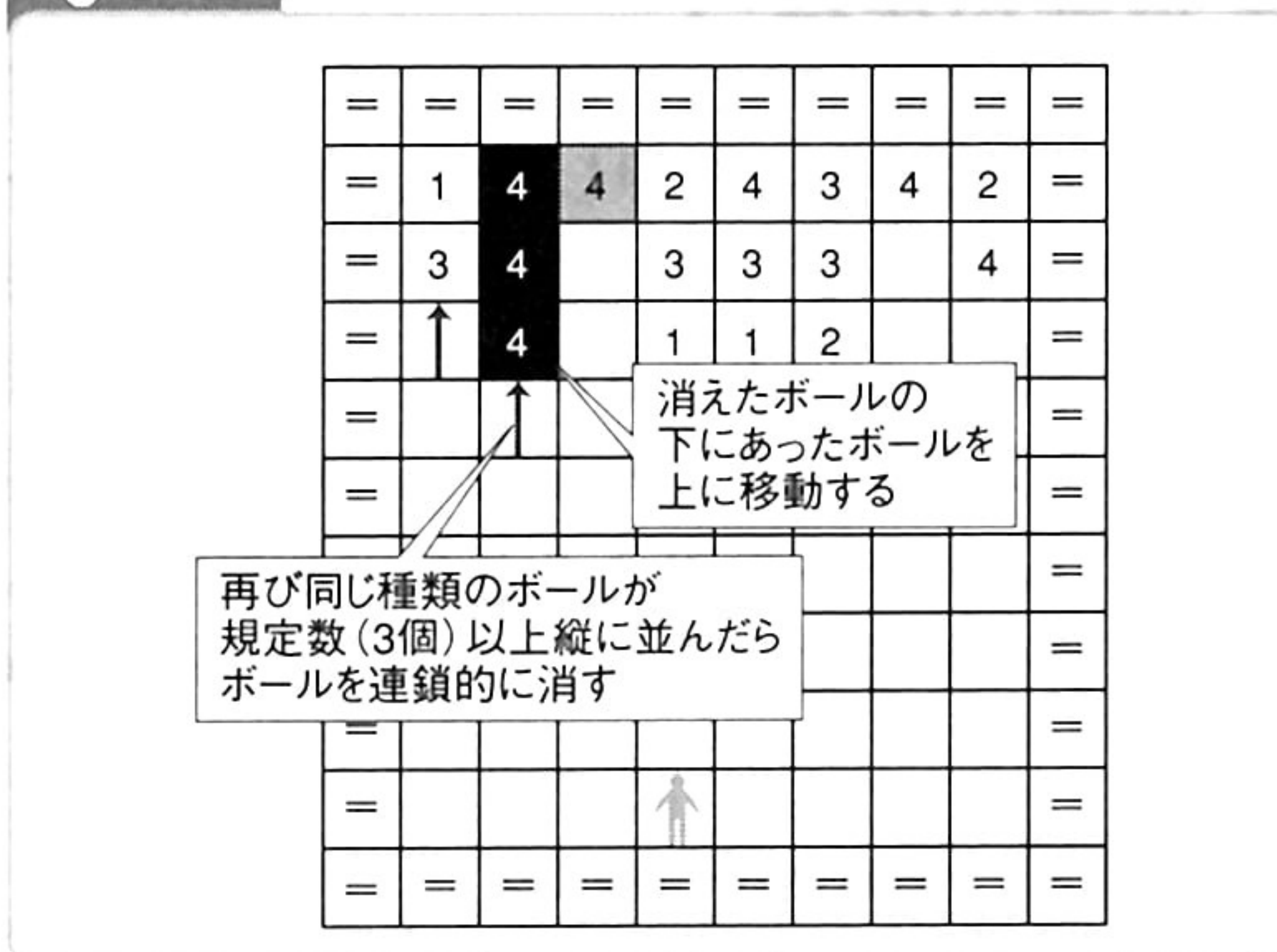



Fig. 5-59 下にあったボールを上に移動する

消えたボールの
下にあったボールを
上に移動する

再び同じ種類のボールが
規定数（3個）以上縦に並んだら
ボールを連鎖的に消す

に書き込み、持っているボールの数を減少させます。

複数のボールを持っているときには、ボールをまとめて戻します。本書のサンプルでは、持っているボールがステージ内に収まりきらないときには、余ったボールがキャラクターの手元に残るようにしています。

ボールを戻したら、ステージ内のすべてのボールについて、同じ種類のボールが縦に規定数（3個）以上並んでいるかどうかを調べます（Fig. 5-58）。並んでいたら、ボールを消します。消えるボールの上下左右に隣接したボールも、一緒に消します。隣接したボールにさらに隣接したボールも、再帰的に消します。

ボールを消したら、下にあったボールを上に移動します（Fig. 5-59）。このとき、再び同じ種類のボールが規定数以上並んだら、ボールを連鎖的に消します。

プログラム

List 5-11はボールを拾って集めるプログラムです。ステージの移動処理、セルがボールかどうかを判定する処理、隣接したボールを消す処理を掲載しました。

移動処理（Move関数）は、入力状態・消去判定状態・消去状態に分かれています。入力状態では、レバー入力に応じて、キャラクターを左右に動かします。ボタン0を押したら、キャラクターの真上にあるボールを拾います。同じ種類のボールが連続していたら、まとめて拾います。ボタン1を押したら、持っているボールをキャラクターの真上に戻します。複数のボールを持っていたら、まとめて戻します。

ボールを戻したら、消去判定状態に移行します。ここでは、ステージ内のすべてのボールについて、同じ種類のボールが縦に規定数（3個）以上並んでいるかどうかを調べます。並んでいたら、それらのボールを消えるボールにします。さらに、隣接したボールを消す処理（Erase関数）を呼び出して、消えるボールの上下左右に隣接したボールを消します。

隣接したボールを消す処理では、上下左右に同じボールが隣接しているかどうかを調べます。

隣接していたら、そのボールの上下左右について、さらに同じボールが隣接しているかどうかを調べます。再帰的に処理を行うことによって、隣接する同じ種類のボールをすべて探し、消えるボールに変えます。

消えるボールがある場合は、消去状態に移行します。ここでは一定時間が経過するのを待ってから、消えるボールを完全に削除します。ボールが消えるまでには、ボールがだんだん薄くなって消える演出を行います。詳細は描画処理（CCollectedBallStageクラスのDraw関数）を参照してください。

セルがボールかどうかを判定する処理（IsBall関数）では、指定されたセルがボールのときにtrueを返します。移動処理や描画処理で共通の処理を使うので、関数にまとめました。

List 5-11　ボールを拾って集める（CCollectedBallStageクラス）

```
// 移動処理
bool CCollectedBallStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 入力状態
    if (State==0) {

        // レバー入力に応じて、キャラクターを左右に動かす
        if (!PrevLever) {
            if (is->Left && CX>1) CX--;
            if (is->Right && CX<xs-1) CX++;
        }
        PrevLever=is->Left||is->Right;

        // ボタンを押したときの処理
        if (!PrevButton) {

            // ボタン0を押したら、ボールを拾う
            if (is->Button[0]) {

                // キャラクターの真上で、
                // 一番近い空ではないセルを探す
                int y;
                for (y=CY-2; Cell->Get(CX, y)==' '; y--) ;

                // まだボールを持っていなかったら、
                // 見つけたセルの種類を、
                // 持っているボールの種類として記録する
                if (BallCount==0) {
                    BallType=Cell->Get(CX, y);
                }

                // 見つけたセルがボールのときの処理
                if (IsBall(BallType)) {
```

```
				//同じ種類のボールが連続しているかぎり、
				//ボールをまとめて拾う
				for (; Cell->Get(CX, y)==BallType; y--) {

					//拾ったボールのセルは空にする
					Cell->Set(CX, y, ' ');

					//持っているボールの数を増やす
					BallCount++;
				}
			}
		}

		//ボタン1を押したら、ボールを戻す
		if (is->Button[1]) {

			//キャラクターの真上で、
			//一番近い空ではないセルを探す
			int y;
			for (y=CY-2; Cell->Get(CX, y)==' '; y--) ;

			//見つけたセルから下に向かって、
			//持っているボールを配置する
			for (y++; y<CY-1 && BallCount>0; y++) {

				//ボールをセルに書き込む
				Cell->Set(CX, y, BallType);

				//持っているボールの数を減らす
				BallCount--;
			}

			//消去判定状態に移行する
			State=1;
		}
	}
	PrevButton=is->Button[0]||is->Button[1];
}

//消去判定状態
if (State==1) {

	//ボールが消えなかった場合には、入力状態に移行する
	State=0;

	//ステージ内のすべてのボールについて調べる
	for (int x=0; x<xs; x++) {
		for (int y=0; y<ys; y++) {
			char c=Cell->Get(x, y);
```

```
                if (IsBall(c)) {

                    // ボールのセルを見つけたら、
                    // 同じ種類のボールが縦に規定数(3個)以上
                    // 並んでいるかどうかを調べる
                    int i;
                    for (i=1; Cell->Get(x, y+i)==c; i++) ;

                    // 規定数以上並んでいたら、
                    // そのボールに隣接する同じ種類のボールを消す
                    if (i>=COLLECTED_BALL_ERASE) {

                        // 隣接する同じ種類のボールを消す
                        Erase(x, y, c);

                        // タイマーを設定して、
                        // 消去状態に移行する
                        Time=0;
                        State=2;
                    }

                    // 消したボールの次のボールから、
                    // 同じ種類のボールが隣接しているかどうかを
                    // 調べる処理を続行する
                    y+=i-1;
                }
            }
        }
    }

    // 消去状態
    if (State==2) {

        // 一定時間が経過するのを待つ
        Time++;
        if (Time==30) {

            // ステージ内から消えるボールを探す
            for (int x=0; x<xs; x++) {
                for (int y=CY-2; y>=0; y--) {

                    // 消えるボールを見つけたときの処理
                    if (Cell->Get(x, y)&0x80) {

                        // 消えるボールよりも下にあるボールを
                        // 上に寄せる
                        for (int i=y; i<CY-2; i++) {
                            Cell->Set(x, i, Cell->Get(x, i+1));
                        }
```

```
                        // ステージ下端のセルは空にする
                        Cell->Set(x, CY-2, ' ');
                    }
                }
            }

            // 入力状態に移行する
            State=0;
        }
    }

    return true;
}

// セルがボールかどうかを判定する処理
bool CCollectedBallStage::IsBall(char c) {

    // セルがボールならば、trueを返す
    return '0'<=c && c<'0'+COLLECTED_BALL_TYPE;
}

// 隣接したボールを消す処理
void CCollectedBallStage::Erase(int x, int y, char c) {

    // 現在のセルが指定された種類のときの処理
    if (Cell->Get(x, y)==c) {

        // 現在のセルに消えるボールのマークを付ける
        Cell->Set(x, y, c|0x80);

        // 隣接する上下左右のセルについても、
        // ボールを消す処理を再帰的に行う
        Erase(x-1, y, c);
        Erase(x+1, y, c);
        Erase(x, y-1, c);
        Erase(x, y+1, c);
    }
}
```

SAMPLE

「COLLECTED BALL」は「ボールを拾って集める」のサンプルです。

レバーの左右（カーソルキーの左右）でキャラクターが動きます。ボタン0（Zキー）を押すと、キャラクターの真上にあるボールを拾うことができます。同じ種類のボールが並んでいるときには、まとめて拾います。

ボタン1（Xキー）を押すと、持っているボールをキャラクターの真上に戻します。複数のボールを持っているときには、ステージ内に収まるかぎり、まとめて戻します。

同じ種類のボールが縦に規定数（3個）以上並ぶと、ボールを消すことができます。ボールが消えると、空いた場所を埋めるように、下にあったボールが上に移動します。再び同じ種類のボールが縦に規定数以上並

ぶと、連鎖的に消えます。

COLLECTED BALL → p. 389

ボールを入れ替える

持っているボールとステージ上のボールを入れ替えることによって、ボールの配置を変えるアクションです。配置を変えることによって、同じ種類のボールを規定数以上隣接させると、ボールを消すことができます。

ステージには多数のボールが積まれています(Fig. 5-60)。レバー操作でカーソルを上下左右に動かすことができます。本書のサンプルでは、カーソルを灰色の矩形で示しています。

カーソルをステージ上のボールに合わせて、ボタンを押すと、ボールを拾うことができます(Fig. 5-61)。ボールがあった場所は空白になります。

ボールを持っているときには、カーソルの位置に、持っているボールを灰色で表示します。この状態で、別のボールにカーソルを重ねてボタンを押すと、持っているボールとステージ上のボールを入れ替えることができます(Fig. 5-62)。次々にボールを入れ替えることで、ステージ内のボールの配置を変えます。

ボールの配置を変えて、同じ種類のボールを規定数(3個)以上隣接させると、ボールを消すことができます(Fig. 5-63)。ボールを消すと、空いた場所を埋めるように、上にあったボールが落下します。ここで再び、同じ種類のボールが規定数以上隣接すると、ボールが連鎖的に消えます。

ボールを入れ替えるアクションを採用したゲームには『対戦とっかえだま』があります。このゲームは『対戦ぱずるだま』によく似ていますが、ボールを落として積むのではなく、ボー

Fig. 5-60 ステージの構成

Fig. 5-61 ボールを拾う

Fig. 5-62 ボールを入れ替える

Fig. 5-63 ボールを消す

ルを入れ替えるため、まったく違ったゲームになっています。「対戦」という名前のとおり、ボールを消すと、相手にボールを送り込むことができます。

アルゴリズム

ボールを入れ替えるアクションを実現するには、ステージをセルで表現します(Fig. 5-64)。ステージの壁は「=」、ボールは「0〜4」の数字で表しました。

レバーを入力したら、カーソルを上下左右に動かします。ボールの上でボタンを押したら、ボールの種類を記録して、セルを空にします(Fig. 5-65)。

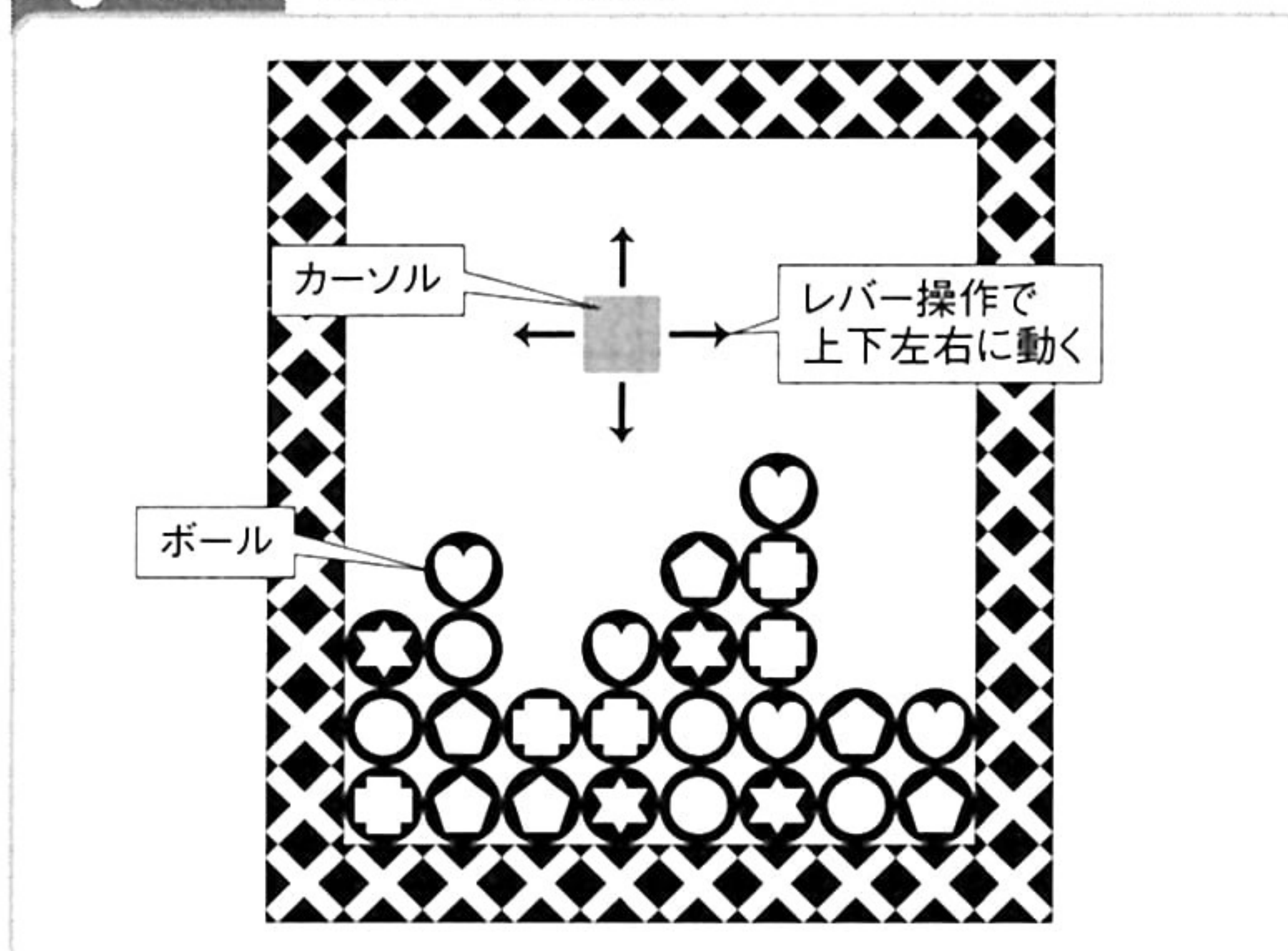
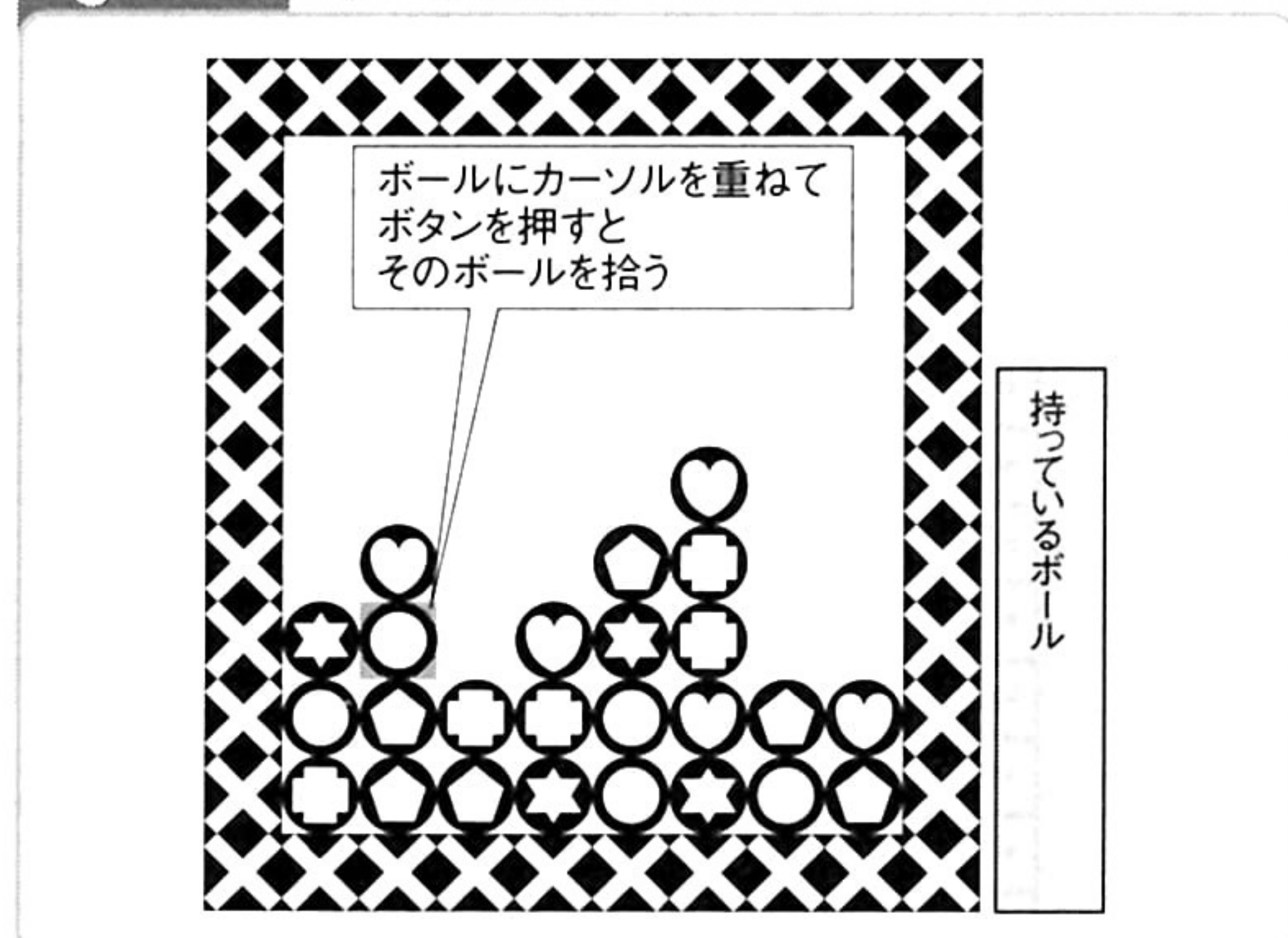
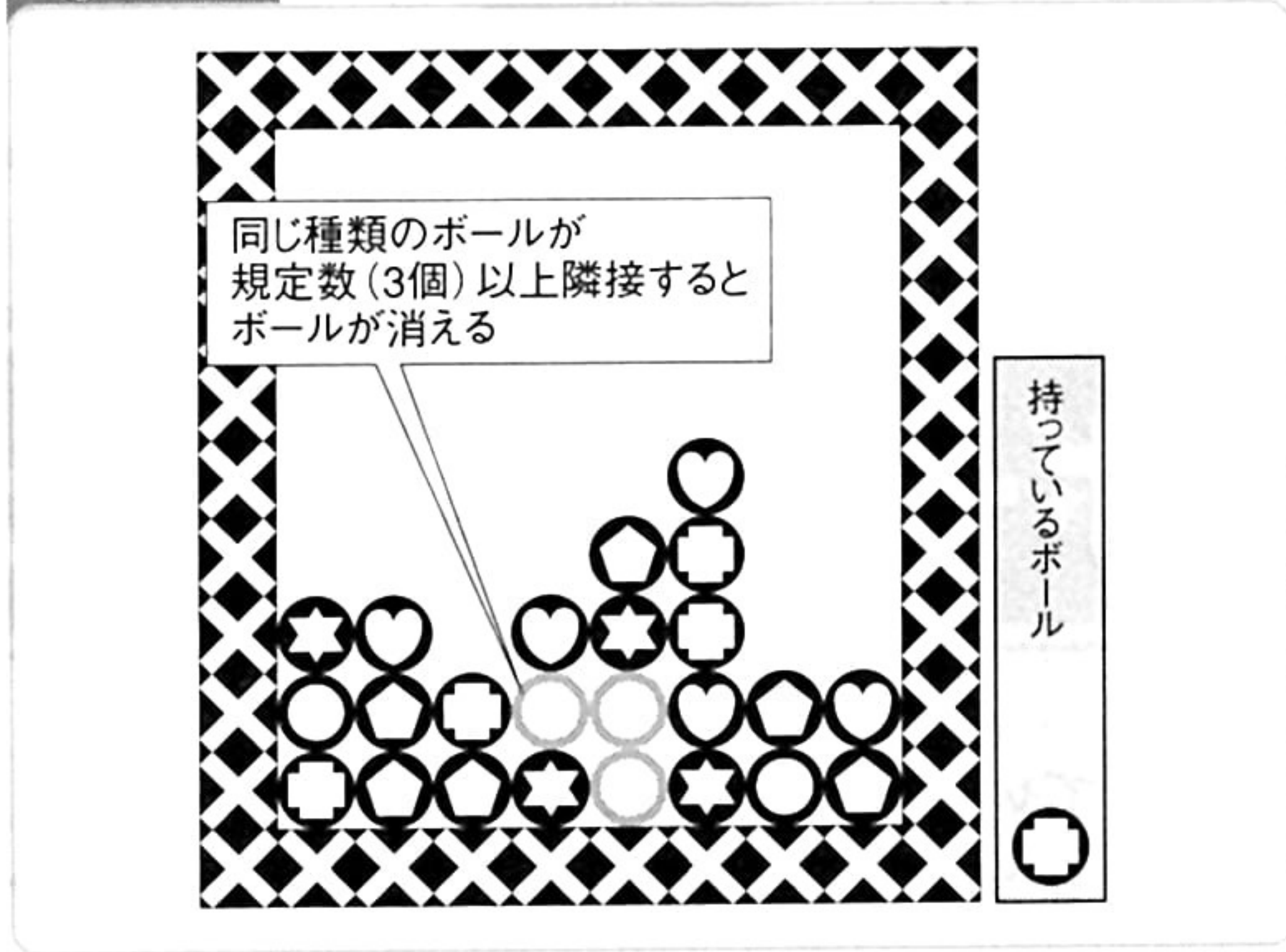
ボールを持っているときに、別のボールの上でボタンを押したら、ボールを入れ替えます(Fig. 5-66)。記録してあるボールの種類をセルに書き込み、別のボールの種類を記録します。

ボールの配置を変えたときには、同じ種類のボールが規定数(3個)以上隣接しているかどうかを調べます(Fig. 5-67)。隣接していたら、それらのボールを消えるボールに変化させます。これは「連鎖的に消す」(→p. 106)と同様の処理です。

Fig. 5-64 ステージをセルで表現する

=	=	=	=	=	=	=	=	=	=
=									=
=									=
=									=
=									=
=						2			=
=		2			4	3			=
=	1	0		2	1	3			=
=	0	4	3	3	0	2	4	2	=
=	3	4	4	1	0	1	0	4	=
=	=	=	=	=	=	=	=	=	=

Fig. 5-65 拾ったボールの種類を記録する

=	=	=	=	=	=	=	=	=	=
=									=
=									=
=									
									=
						2			=
=		2			4	3			=
=	1	0		2	1	3			=
=	0	4	3	3	0	2	4	2	=
=	3	4	4	1	0	1	0	4	=
=	=	=	=	=	=	=	=	=	=

Fig. 5-66　ボールの種類を入れ替える

ボタンを押したら
持っているボールと
カーソル位置のボールを
入れ替える

ボールの種類を
入れ替える

持っているボール

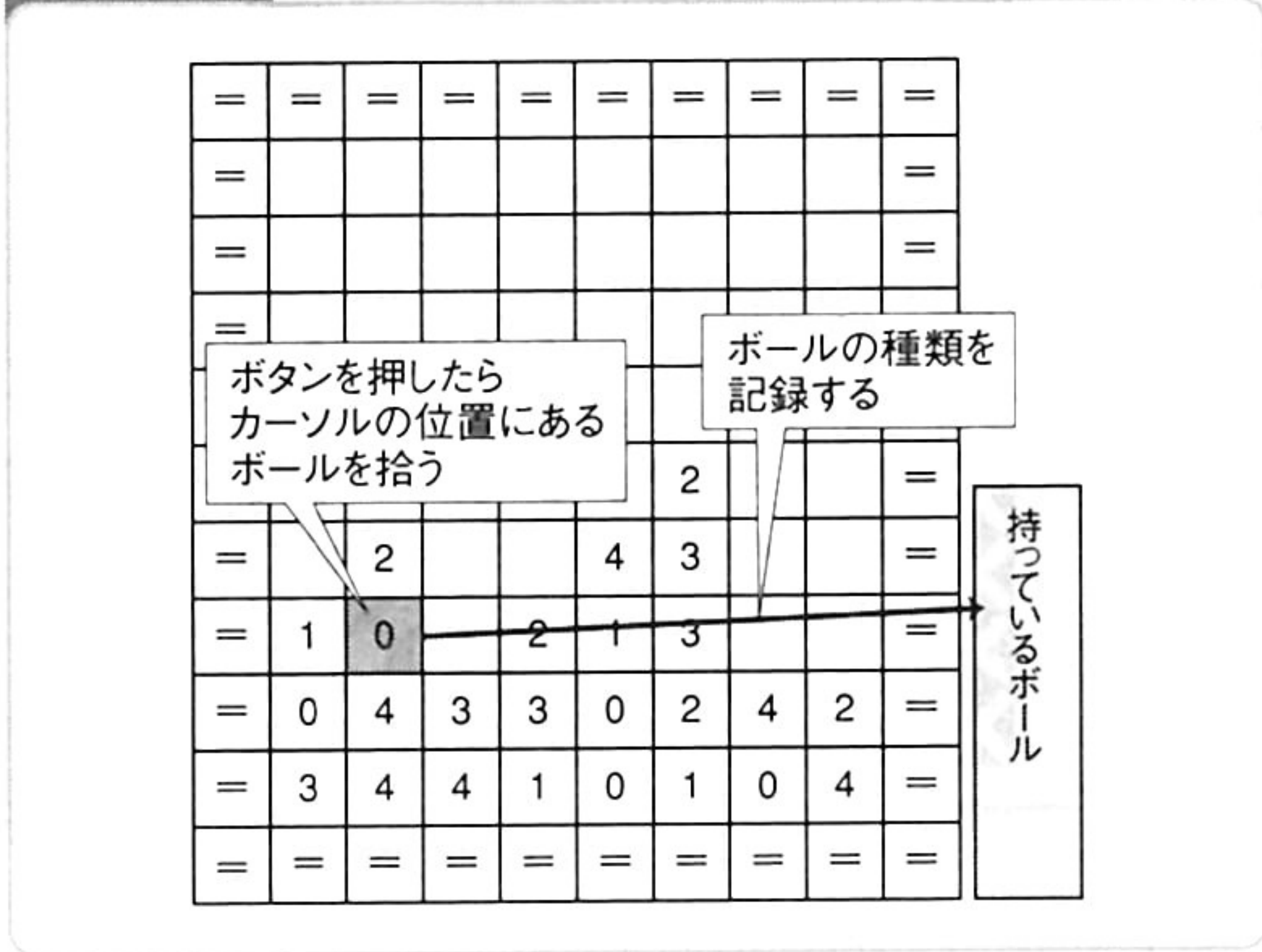
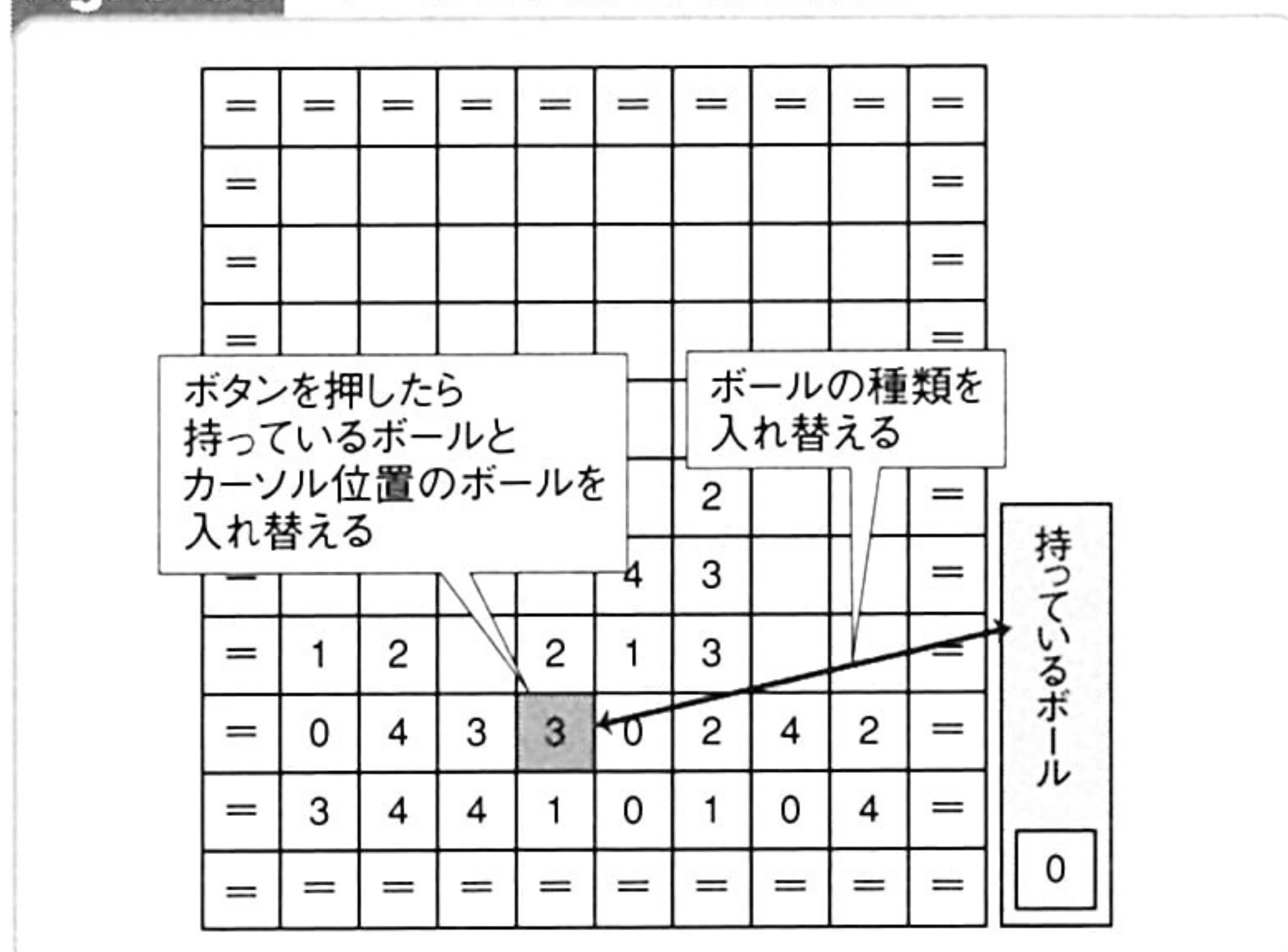



Fig. 5-67　ボールが規定数以上隣接したかを調べる

同じ種類のボールが
規定数（3個）以上隣接したら
ボールを消す

持っているボール

一定時間が経過したら、消えるボールを空のセルに変えて、完全に消します。一定時間が経過するまでは、ボールをだんだん薄く表示するなどして、消える様子を表現します。

ボールを完全に消したら、上にあったボールを落下させます。このとき、再び同じ種類のボールが規定数以上隣接したら、連鎖的に消します。

プログラム

List 5-12はボールを入れ替えるプログラムです。ステージの移動処理、セルがボールかどうかを判定する処理、隣接したボールを消す処理を掲載しました。

移動処理（Move関数）は入力状態・落下状態・消去判定状態・消去状態に分かれています。入力状態では、レバー入力に応じて、カーソルを上下左右に動かします。ボタンを押したら、持っているボールと、カーソルの位置にあるボールとを入れ替えます。ボールを入れ替えたら、落下状態に移行します。

落下状態では、ステージ内の空のセルを探して、上にあるボールを落下させます。次に、消去判定状態に移行して、同じ種類のボールが規定数以上隣接しているかどうかを調べます。隣接していたら、それらのボールを消えるボールにして、消去状態に移行します。

消去状態では、一定時間が経過するのを待ってから、ボールを完全に消します。ステージの描画処理（CSwappedBallStageクラスのDraw関数）では、ボールが完全に消える前に、ボールがだんだん薄くなって消えるような表示を行います。

セルがボールかどうかを判定する処理（IsBall関数）は、指定されたセルがボールのときにtrueを返します。この処理は移動処理や描画処理で使います。

隣接したボールを消す処理（Erase関数）は、隣接した同じ種類のボールを数える処理と、隣接したボールを消えるボールに変える処理を兼ねています。上下左右に同じ種類のセルが隣接していたら、隣接したセルの上下左右も再帰的に調べることによって、隣接した同じ種類のボールをすべて探し出します。

List 5-12 ボールを入れ替える(CSwappedBallStageクラス)

```
// 移動処理
bool CSwappedBallStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 入力状態
    if (State==0) {

        // レバー入力に応じて、カーソルを上下左右に動かす
        if (!PrevLever) {
            int cx=CX, cy=CY;
            if (is->Left) cx--; else
            if (is->Right) cx++; else
            if (is->Up) cy--; else
            if (is->Down) cy++;
            if (Cell->Get(cx, cy)!='=') {
                CX=cx;
                CY=cy;
            }
        }
        PrevLever=is->Left||is->Right||is->Up||is->Down;

        // ボタンを押したら、持っているボールと、
        // カーソルの位置にあるボールを入れ替える
        if (!PrevButton && is->Button[0]) {

            // 持っているボールの種類と、
            // カーソルの位置にあるボールの種類を入れ替える
            char c=Cell->Get(CX, CY);
            Cell->Set(CX, CY, BallType);
            BallType=c;

            // 落下状態に移行する
            State=1;
        }
        PrevButton=is->Button[0];
    }

    // 落下状態
    if (State==1) {

        // ステージ内のすべてのボールについて、
        // 下に空のセルがあるボールを落下させる
        for (int x=0; x<xs; x++) {
            for (int y=0; y<ys; y++) {

                // 空のセルを見つけたときの処理
                if (Cell->Get(x, y)==' ') {
```

```
                // 上にあるボールを落下させる
                for (int i=y; i>2; i--) {
                    Cell->Set(x, i, Cell->Get(x, i-1));
                }

                // ステージ上端は空のセルにする
                Cell->Set(x, 2, ' ');
            }
        }
    }

    // 消去判定状態に移行する
    State=2;
}

// 消去判定状態
if (State==2) {

    // ボールが消えなかったときには、入力状態に移行する
    State=0;

    // ステージ内のすべてのボールについて調べる
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {

            // 同じ種類のボールが規定数(3個)以上隣接していたら、
            // それらのボールを消す
            char c=Cell->Get(x, y);
            if (
                IsBall(c) &&
                Erase(x, y, c, 0x80)>=SWAPPED_BALL_ERASE
            ) {
                // ボールを消えるボールにする
                Erase(x, y, c|0x80, 0x40);

                // タイマーを設定して、消去状態に移行する
                Time=0;
                State=3;
            }
        }
    }

    // ボールのカウントずみのマークを解除する
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {
            Cell->Set(x, y, Cell->Get(x, y)&0x7f);
        }
    }
}
```

```
    // 消去状態
    if (State==3) {

        // 一定時間が経過するのを待つ
        Time++;
        if (Time==30) {

            // 消えるボールを完全に削除する
            for (int x=0; x<xs; x++) {
                for (int y=0; y<ys; y++) {

                    // 消えるボールを空のセルにする
                    if (Cell->Get(x, y)&0x40) {
                        Cell->Set(x, y, ' ');
                    }
                }
            }

            // 落下状態に移行する
            State=1;
        }
    }

    return true;
}

// セルがボールかどうかを判定する処理
bool CSwappedBallStage::IsBall(char c) {
    return '0'<=c && c<'0'+SWAPPED_BALL_TYPE;
}

// 隣接したボールを消す処理
int CSwappedBallStage::Erase(int x, int y, char c, int mask) {

    // 現在のセルが指定された種類かどうかを調べる
    if (Cell->Get(x, y)==c) {

        // 指定された種類ならば、セルにマークを付ける
        // (特定のビットを1にする)
        Cell->Set(x, y, c|mask);

        // 上下左右に隣接するセルについても再帰的に調べて、
        // 同じ種類のセルが隣接する個数の合計を返す
        return
            1+
            Erase(x-1, y, c, mask)+
            Erase(x+1, y, c, mask)+
            Erase(x, y-1, c, mask)+
            Erase(x, y+1, c, mask);
```

```
        }

        // 現在のセルが指定された種類でなければ、
        // 同じ種類のセルが隣接する個数として0を返す
        return 0;
    }
```

SAMPLE

「SWAPPED BALL」は「ボールを入れ替える」のサンプルです。

レバーの上下左右（カーソルキーの上下左右）でカーソルが動きます。ボタン0（Zキー）を押すと、カーソルの位置にあるボールを拾います。ボールを持っているときにボタンを押すと、持っているボールと、カーソルの位置にあるボールとを入れ替えます。

ボールを入れ替えたときに、同じ種類のボールが規定数（3個）以上隣接すると、ボールが消えます。ボールが消えて空いた場所には、上にあったボールが落ちてきます。ここで再び同じ種類のボールが並ぶと、連鎖的にボールを消すことができます。

SWAPPED BALL → p. 389

ボールをヘビ状に動かす

ボールがヘビ状に連なって出現するアクションです。レバー操作で先頭のボールを動かすと、ヘビが動くように、残りのボールがついてきます。

ステージにはボールが積まれています（Fig. 5-68）。ステージ上方にはボールがヘビ状に連なって出現します。レバーを入力すると、先頭のボールを上下左右に動かすことができます（Fig. 5-69）。先頭以外のボールは、まるでヘビのような動きで先頭のボールを追いかけます。

Fig. 5-68 ステージの構成

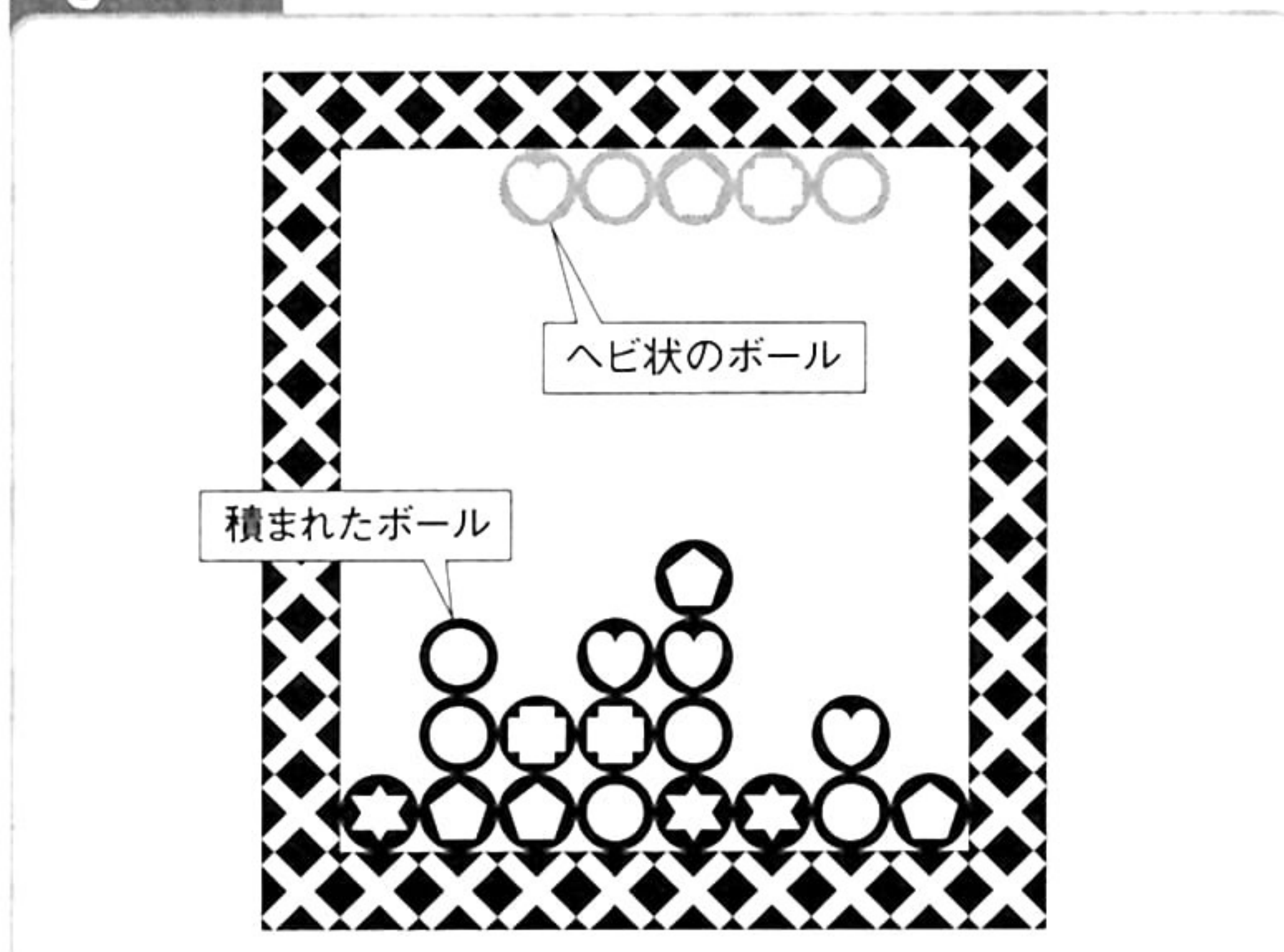



Fig. 5-69 ボールを動かす

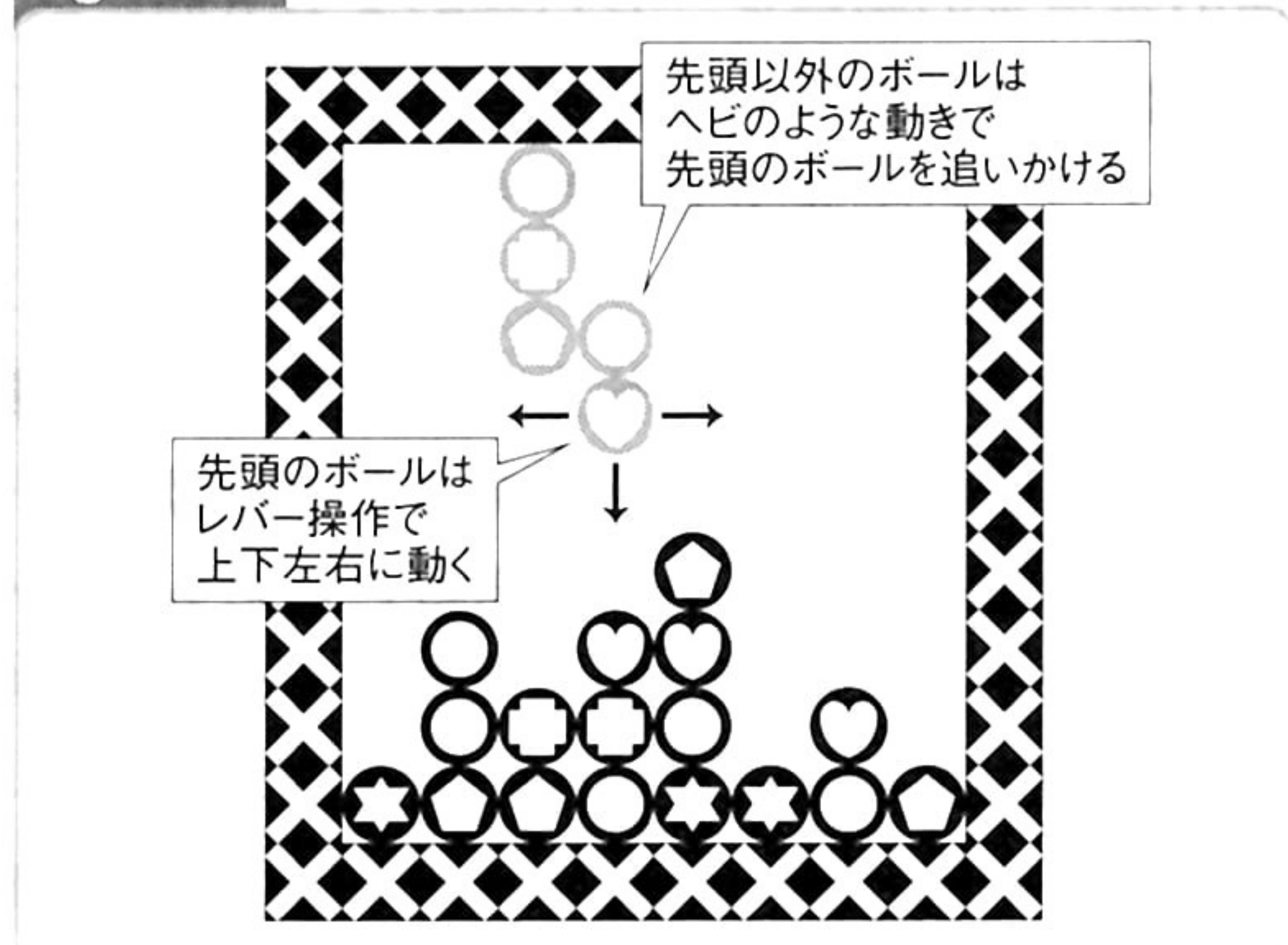



Fig. 5-70 ボールを固定する

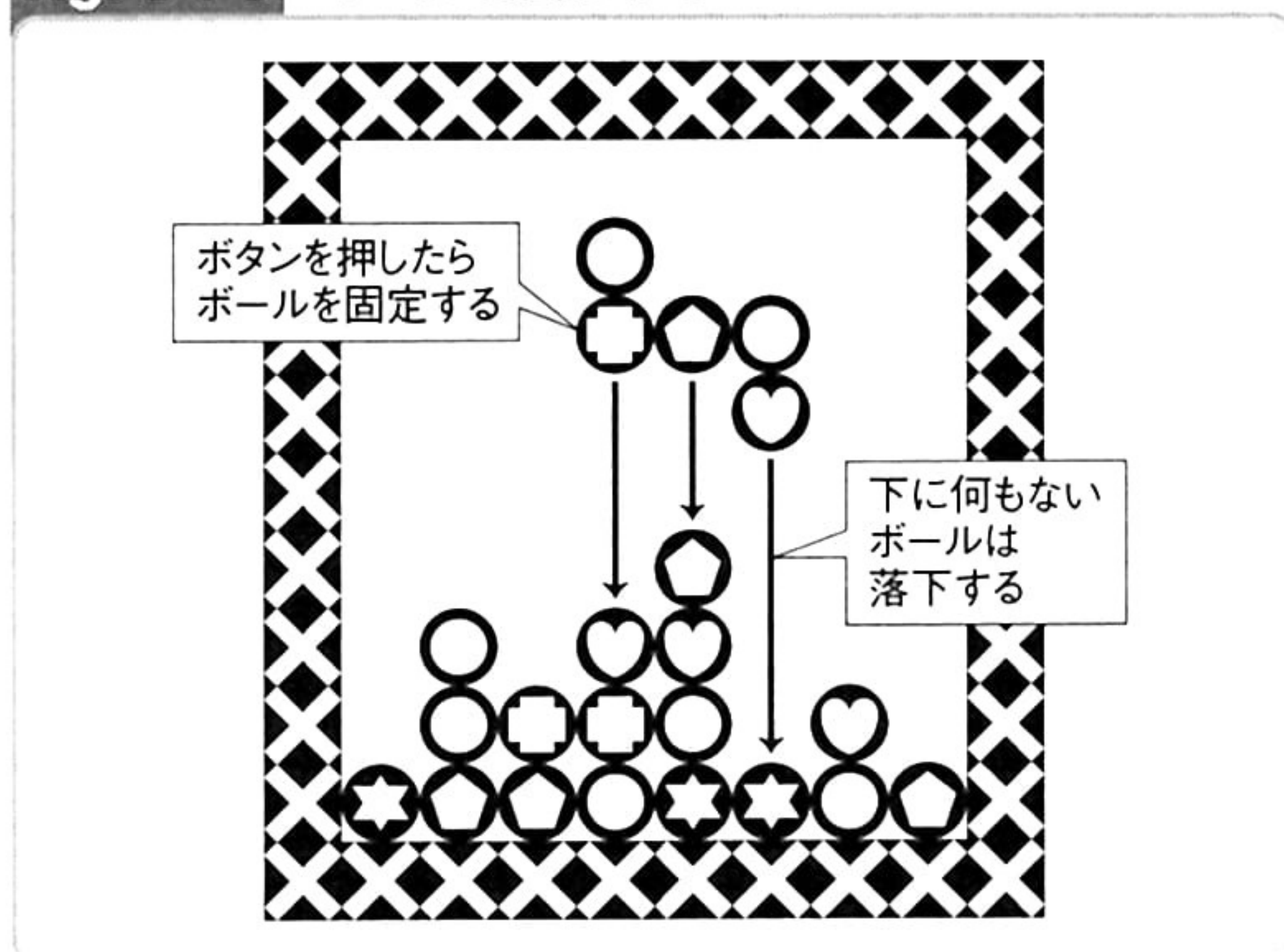
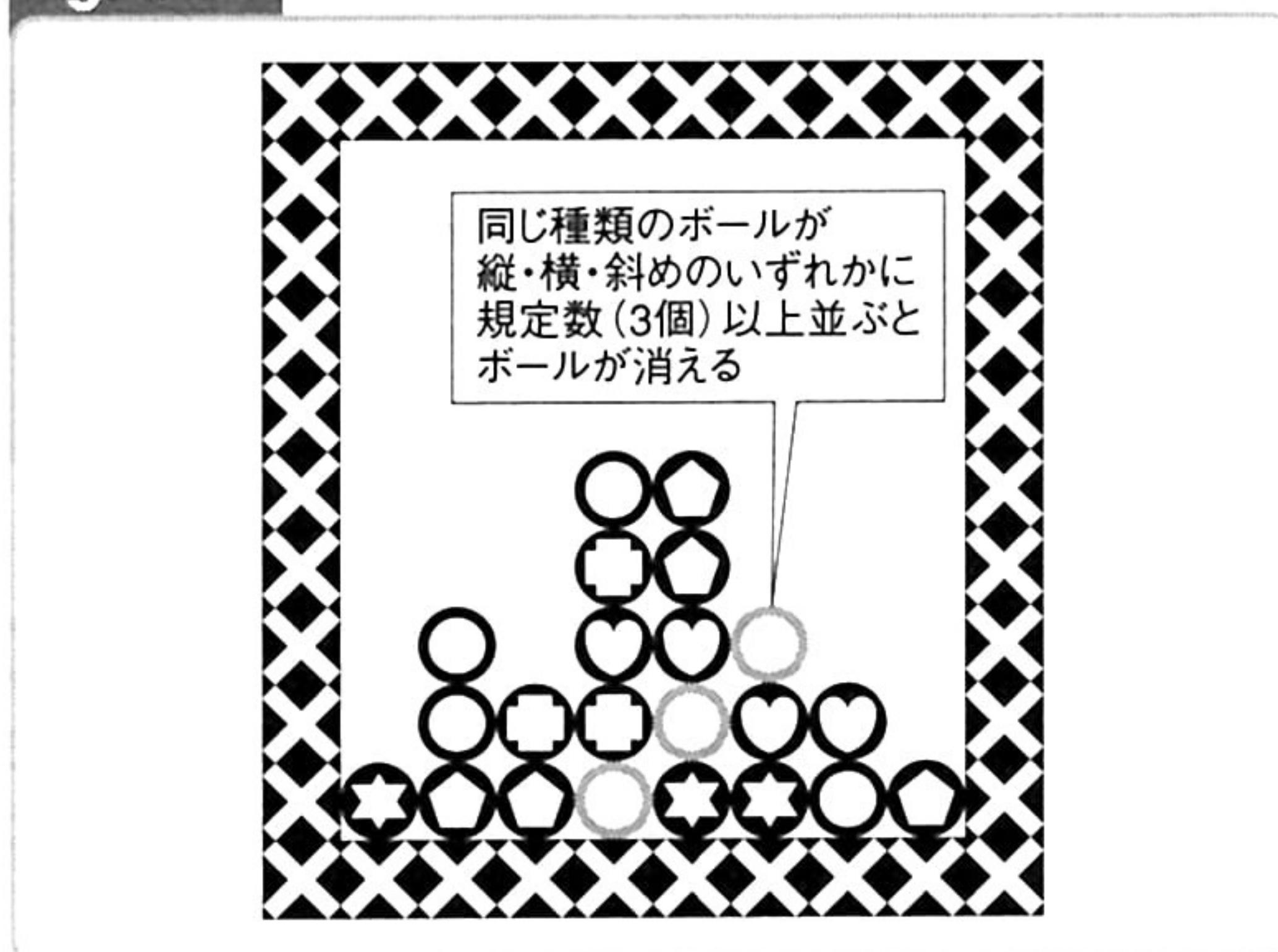



Fig. 5-71 ボールが消える

ボタンを押すと、ヘビ状に連なったボールは、その場所に固定されます(Fig. 5-70)。そして、固定されたボールの下に何もないときには、ボールは落下します。

ボールが着地したときに、同じ種類のボールが縦・横・斜めのいずれかに規定数(3個)以上並んでいたら、ボールが消えます(Fig. 5-71)。ボールが消えると、上にあったボールは落下します。このとき、再び同じ種類のボールが規定数以上並ぶと、連鎖的に消えます。

ボールをヘビ状に動かすアクションは『オーマイガァー!』に採用されています。このゲームでは、さまざまな長さのヘビ状のボールが、ステージ上方に出現します。レバーで、先頭のボールを上下左右に動かすことができます。残りのボールは先頭のボールを追いかけるように動きます。本書のサンプルとは異なり、ボールは時間とともに少しずつ落下します。他のボールの上に着地すると、ボールは固定されて、次のボールがステージ上方に出現します。

アルゴリズム

ボールをヘビ状に動かすには、それぞれのボールの座標を記録しておきます(Fig. 5-72)。レバーを入力したら、先頭以外のボールの座標を、1つ前のボールの座標にします。そして、先頭のボールの座標を、レバーの入力方向に応じた新しい座標にします。これで、先頭以外のボールは、先頭のボールを追いかけるように動きます。

ステージはセルで表現します(Fig. 5-73)。ステージの壁は「=」、ボールは「0〜4」の数値で表しました。

先頭のボールを動かすときには、セルを調べて、移動先が空のセルのときだけ動かします。移動先に壁や他のボールがあるときには、ボールを動かしません。また、先頭のボールの移動先に、先頭以外のボールがあるときにも、ボールを動かすことはできません。

ボタンを押したら、セルにボールの種類を書き込んで、現在の場所にヘビ状のボールを固定します(Fig. 5-74)。ボールの下が空のセルならば、ボールを落下させます。

ボールが縦・横・斜めに規定数(3個)以上並んだら、ボールを消します(Fig. 5-75)。ボール

Fig. 5-72　各ボールの座標を記録する

先頭のボール

<ボール0>	<ボール1>	<ボール2>	<ボール3>	<ボール4>
種類2	種類0	種類4	種類3	種類0
座標(3,1)	座標(4,1)	座標(5,1)	座標(6,1)	座標(7,1)

先頭以外のボールの座標を
1つ前のボールの座標にする

<ボール0>	<ボール1>	<ボール2>	<ボール3>	<ボール4>
種類2	種類0	種類4	種類3	種類0
座標(3,1) →	座標(3,1) →	座標(4,1) →	座標(5,1) →	座標(6,1)

先頭のボールの座標をレバーの
入力方向に応じた新しい座標にする

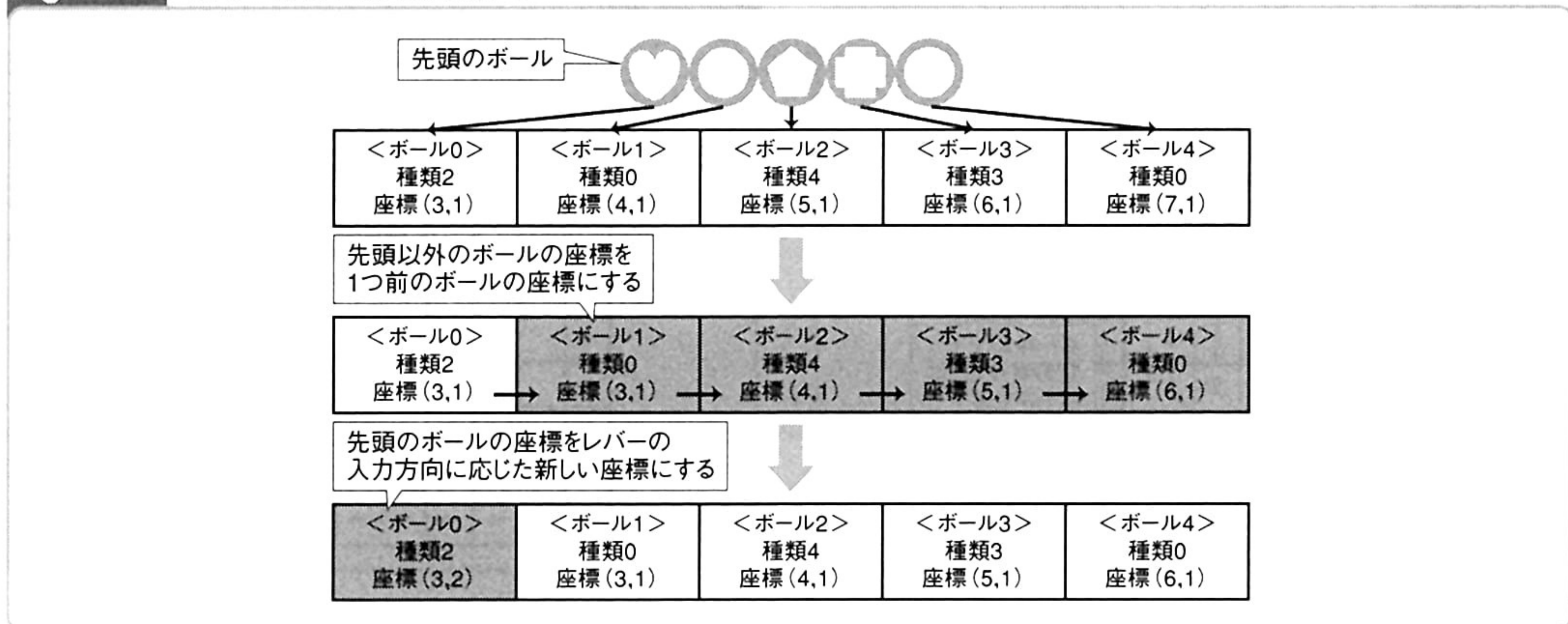
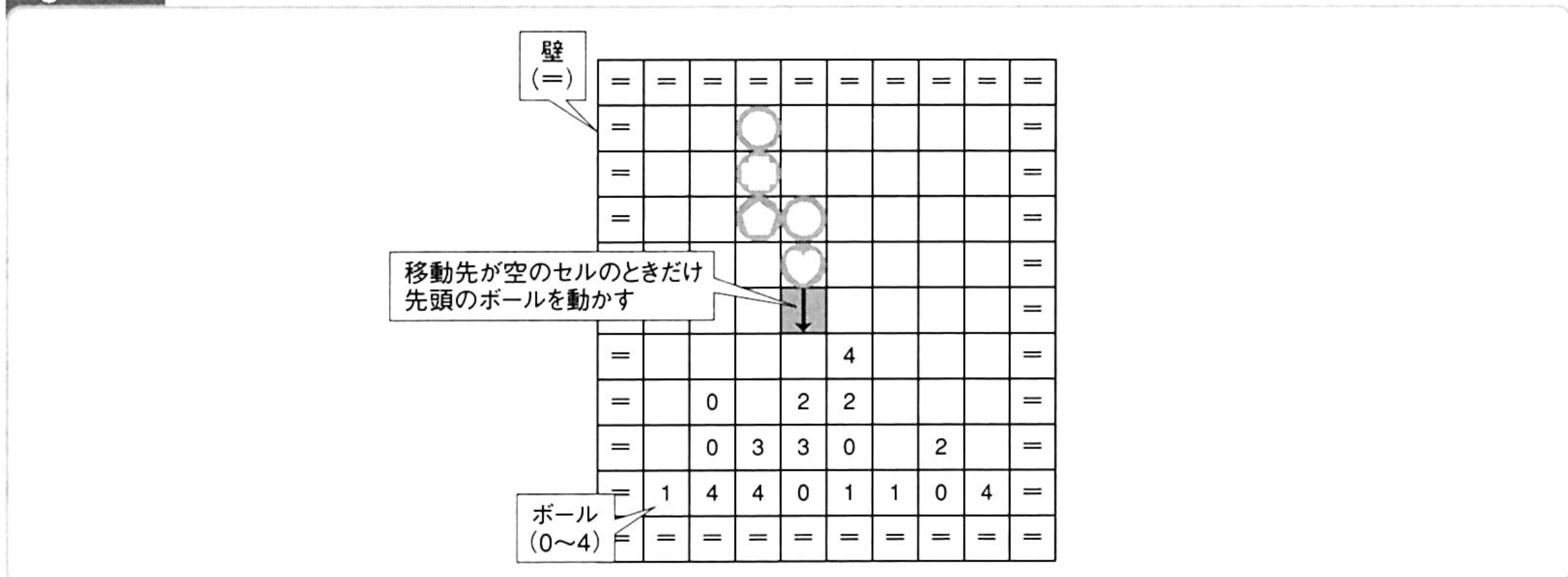
<ボール0>	<ボール1>	<ボール2>	<ボール3>	<ボール4>
種類2	種類0	種類4	種類3	種類0
座標(3,2)	座標(3,1)	座標(4,1)	座標(5,1)	座標(6,1)

Fig. 5-73　ステージをセルで表現する

壁
(＝)

移動先が空のセルのときだけ
先頭のボールを動かす

ボール
(0〜4)

＝	＝	＝	＝	＝	＝	＝	＝	＝	＝
＝									＝
＝									＝
＝									＝
＝									＝
＝									＝
＝					4				＝
＝		0		2	2				＝
＝		0	3	3	0		2		＝
＝	1	4	4	0	1	1	0	4	＝
＝	＝	＝	＝	＝	＝	＝	＝	＝	＝

を消えるボールに変化させて、一定時間が経過したら、完全に消します。

ボールが消えて空いた場所には、上にあったボールを落下させます。ここで再びボールが規定数以上並んだら、連鎖的にボールを消します。

ボールが並んだかどうかを調べる方法は、「縦横斜めに揃える」(→p. 82) や「落ちてくるブロックを拾って積む」(→p. 262) と同様です。下・右・左下・右下について、同じ種類のボールが規定数以上並んでいるかどうかを調べます。上・左・右上・左上については、下・右・左下・右下と重複するので、調べる必要はありません。

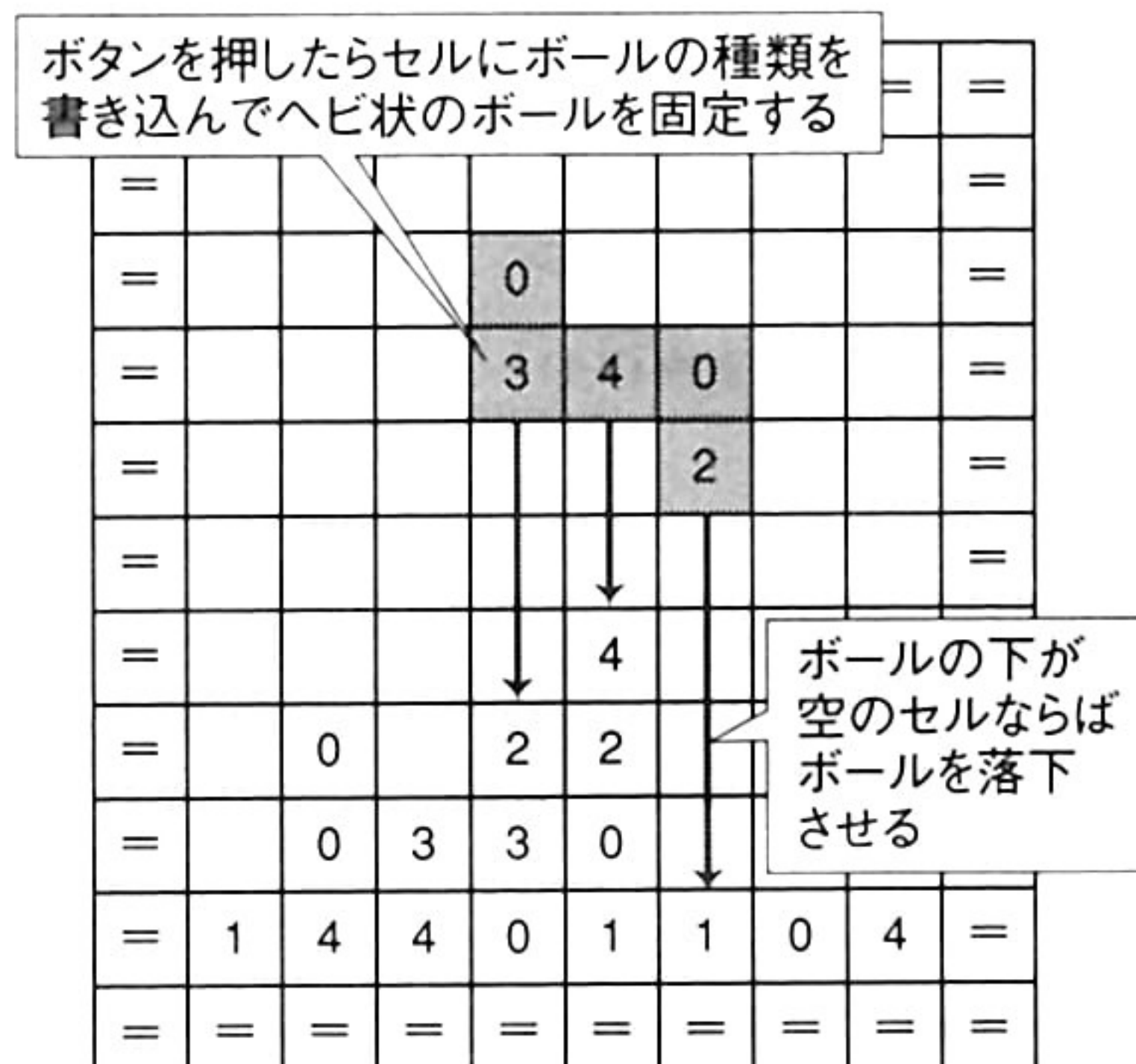
プログラム

List 5-13はボールをヘビ状に動かすプログラムです。ステージの移動処理と、セルがボールかどうかを判定する処理を掲載しました。

Fig. 5-74 セルにボールの種類を書き込む

								=	=
=									=
=				0					=
=				3	4	0			=
=						2			=
=									=
=					4				
=		0		2	2				
=		0	3	3	0				
=	1	4	4	0	1	1	0	4	=
=	=	=	=	=	=	=	=	=	=

Fig. 5-75 ボールを消す

=	=	=	=	=	=	=	=	=	=
=									=
=									
=									
=									
=				0	4				=
=				3	4				=
=		0		2	2	0			=
=		0	3	3	0	2	2		=
=	1	4	4	0	1	1	0	4	=
=	=	=	=	=	=	=	=	=	=

移動処理（Move関数）は、初期状態・入力状態・落下状態・消去判定状態・消去状態に分かれています。初期状態では、ヘビ状のボールの座標と種類を初期化してから、入力状態に移行します。ボールはステージ上方に横一列に並べ、ボールの種類はランダムに選びます。

入力状態では、レバーの入力に応じて、先頭のボールを上下左右に動かします。先頭以外のボールは、1つ前のボールの座標に移動することによって、先頭のボールを追いかけるように動かします。ボタンを押したら、その場所にボールを固定して、落下状態に移行します。

落下状態では、ステージ内の空のセルについて、上にあるセルを落下させます。落下させたら、消去判定状態に移行します。

消去判定状態では、同じ種類のボールが縦・横・斜めのいずれかに規定数（3個）以上並んだかどうかを調べます。ボールが並んだら、それらのボールを消えるボールに変化させて、消去状態に移行します。消去状態では、一定時間が経過するのを待ってから、消えるボールを空のセルにすることによって、完全に消去します。

セルがボールかどうかを判定する処理（IsBall関数）は、セルがボールのときにtrueを返します。この処理は移動処理や描画処理で使います。

List 5-13 ボールをヘビ状に動かす（CSnakeBallStageクラス）

```
// 移動処理
bool CSnakeBallStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 初期状態
    if (State==0) {

        // 入力状態に移行する
        State=1;
```

```
        // ヘビ状のボールの座標と種類を初期化する
        for (int i=0; i<SNAKE_BALL_COUNT; i++) {

            // ボールが横1列に並ぶように、座標を設定する
            CX[i]=(xs-SNAKE_BALL_COUNT)/2+i;
            CY[i]=2;

            // ボールの種類はランダムに選ぶ
            BallType[i]='0'+Rand.Int31()%SNAKE_BALL_TYPE;

            // ボールの出現位置に他のボールがある場合は、
            // 初期状態を維持して、ボールを出現させない
            if (Cell->Get(CX[i], CY[i])!=' ') {
                State=0;
            }
        }
    }

    // 入力状態
    if (State==1) {

        // レバーの入力に応じて、先頭のボールを上下左右に動かす
        if (!PrevLever) {

            // レバーの入力方向に従って、移動先の座標を求める
            int cx=CX[0], cy=CY[0];
            if (is->Left) cx--; else
            if (is->Right) cx++; else
            if (is->Up) cy--; else
            if (is->Down) cy++;

            // 先頭のボールの移動先が空のセルならば、ボールを移動する
            if (Cell->Get(cx, cy)==' ') {

                // 先頭のボールが追随するボールに接触するかどうかを調べる
                int i;
                for (i=0; i<SNAKE_BALL_COUNT; i++) {
                    if (CX[i]==cx && CY[i]==cy) break;
                }

                // 接触しない場合には、ボールを移動する
                if (i==SNAKE_BALL_COUNT) {

                    // 追随するボールの座標を1つ前のボールの座標にする
                    for (i=SNAKE_BALL_COUNT-1; i>0; i--) {
                        CX[i]=CX[i-1];
                        CY[i]=CY[i-1];
                    }

                    // 先頭のボールを動かす
```

```
                    CX[0]=cx;
                    CY[0]=cy;
                }
            }
        }
        PrevLever=is->Left||is->Right||is->Up||is->Down;

        // ボタンを押したら、その場所にボールを固定する
        if (!PrevButton && is->Button[0]) {

            // ボールの種類をセルに書き込む
            for (int i=0; i<SNAKE_BALL_COUNT; i++) {
                Cell->Set(CX[i], CY[i], BallType[i]);
            }

            // 落下状態に移行する
            State=2;
        }
        PrevButton=is->Button[0];
    }

    // 落下状態
    if (State==2) {

    // ステージ内の空のセルを探す
        for (int x=0; x<xs; x++) {
            for (int y=0; y<ys; y++) {

                // 空のセルが見つかったときの処理
                if (Cell->Get(x, y)==' ') {

                    // 空のセルの上にあるセルを1段ずつ落下させる
                    for (int i=y; i>2; i--) {
                        Cell->Set(x, i, Cell->Get(x, i-1));
                    }

                    // ステージ上端のセルは空にする
                    Cell->Set(x, 2, ' ');
                }
            }
        }

    // 消去判定状態に移行する
        State=3;
    }

    // 消去判定状態
    if (State==3) {

        // ボールが消えなかったら、初期状態に移行する
```

```
State=0;

// ステージ内のすべてのボールについて調べる
for (int y=0; y<ys; y++) {
    for (int x=0; x<xs; x++) {

        // ボールのセルについて、
        // 縦・横・斜めに同じ種類のボールが
        // 並んでいるかどうかを調べる
        char c=Cell->Get(x, y);
        if (IsBall(c&0x7f)) {

            // 下・右・左下・右下について調べる
            static const int
                vx[]={0, 1, -1, 1},
                vy[]={1, 0, 1, 1};
            for (int v=0; v<4; v++) {

                // 同じ種類のボールが並んでいる個数を数える
                int count=0;
                for (
                    int i=x, j=y;
                    (Cell->Get(i, j)&0x7f)==c;
                    i+=vx[v], j+=vy[v]
                ) {
                    count++;
                }

                // 規定数(3個)以上のボールが並んでいたら、
                // ボールを消えるボールにする
                if (count>=SNAKE_BALL_ERASE) {
                    for (
                        int i=x, j=y, k=0;
                        k<count;
                        i+=vx[v], j+=vy[v], k++
                    ) {
                        // 消えるボールのマークを付ける
                        Cell->Set(i, j, c|0x80);
                    }

                    // タイマーを設定し、消去状態に移行する
                    Time=0;
                    State=4;
                }
            }
        }
    }
}
}
```

```
    // 消去状態
    if (State==4) {

    // 一定時間が経過するのを待ってから、
    // 消えるボールを完全に消す
        Time++;
        if (Time==30) {

            // 消えるボールを探す
            for (int x=0; x<xs; x++) {
                for (int y=0; y<ys; y++) {

                    // 消えるボールを空のセルにする
                    if (Cell->Get(x, y)&0x80) {
                        Cell->Set(x, y, ' ');
                    }
                }
            }

            // 落下状態に移行する
            State=2;
        }
    }

    return true;
}

// セルがボールかどうかを判定する処理
bool CSnakeBallStage::IsBall(char c) {
    return '0'<=c && c<'0'+SNAKE_BALL_TYPE;
}
```

SAMPLE

「SNAKE BALL」は「ボールをヘビ状に動かす」のサンプルです。

ステージの上方に、ヘビ状に連なったボールが出現します。レバーの上下左右(カーソルキーの上下左右)で、先頭のボールが動きます。先頭以外のボールは、先頭のボールを追いかけるように動きます。

ボタン0(Zキー)を押すと、その場所にボールを固定します。ボールの下に他のボールがなければ、ボールは落下します。

同じ種類のボールを縦・横・斜めのいずれかに規定数(3個)以上並べると、ボールを消すことができます。ボールを消すと、空いた場所の上にあるボールが落ちてきます。ここで再び同じ種類のボールが並ぶと、連鎖的に消えます。

SNAKE BALL → p. 389

ばねでボールを撃つ

ばねの力でボールを弾いて撃つアクションです。ばねを引く強さに応じて、ボールの飛距離が変わります。

ステージにはボールが積まれています (Fig. 5-76)。ステージの左上にあるボールは、これから撃つボールです。

レバーを下に入力すると、撃つボールが下がります (Fig. 5-77)。これはばねを引いている状態です。レバーを下に入れ続けると、画面の下端に達するまで、ボールは少しずつ下がっていきます。

レバーを中央に戻すと、ボールを撃つことができます (Fig. 5-78)。ばねを引いた強さに応じて、ボールの飛距離が変わります。ばねを強く引くほど、つまりボールを下げてから撃つほど、ボールは遠くに飛びます。

撃ったボールは、ボールが積まれた領域に落ちていきます (Fig. 5-79)。ばねを弱く引くと左側 (近く) に、強く引くと右側 (遠く) に落ちます。

ボールが着地したときに、同じ種類のボールが規定数 (3個) 以上隣接すると、ボールは消えます (Fig. 5-80)。ボールが消えて空いた場所には、上にあったボールが落ちてきます。ここで再び同じ種類のボールが隣接すると、連鎖的に消えます。

ばねでボールを撃つアクションは『魚ポコ』に採用されています。このゲームでは、「ぶら下がったボール」(→p. 301) のように、ボールが互い違いに積まれています。着地したボールは、互い違いに積まれたボールの上を、左右に転がりながら落ちていきます。ばねでボールを撃つ動きや、ボールが転がる動きが面白いので、見た目にも楽しいゲームです。レバーの下方向だけを使う操作方法もユニークです。

Fig. 5-76 ステージの構成

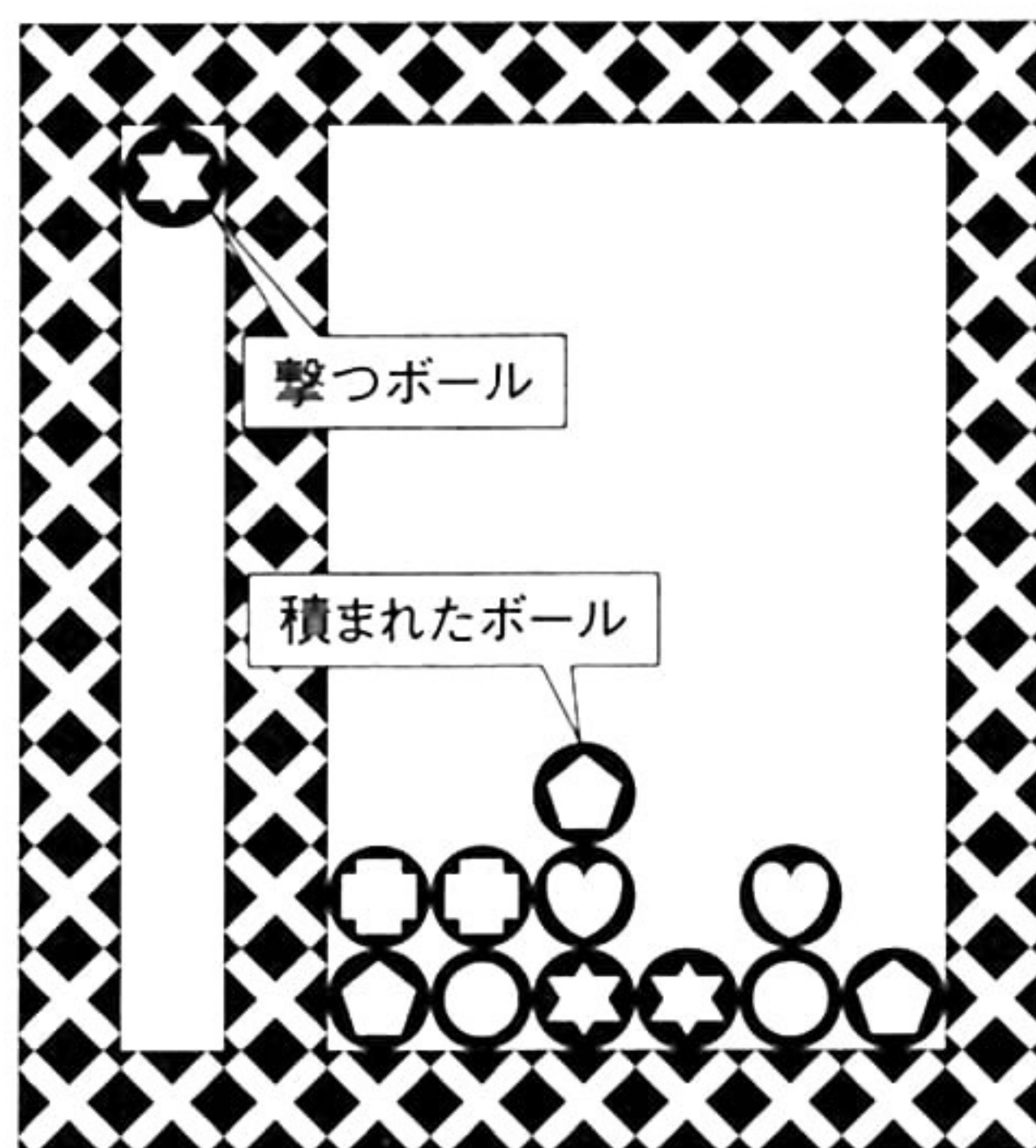



Fig. 5-77 ばねを引く

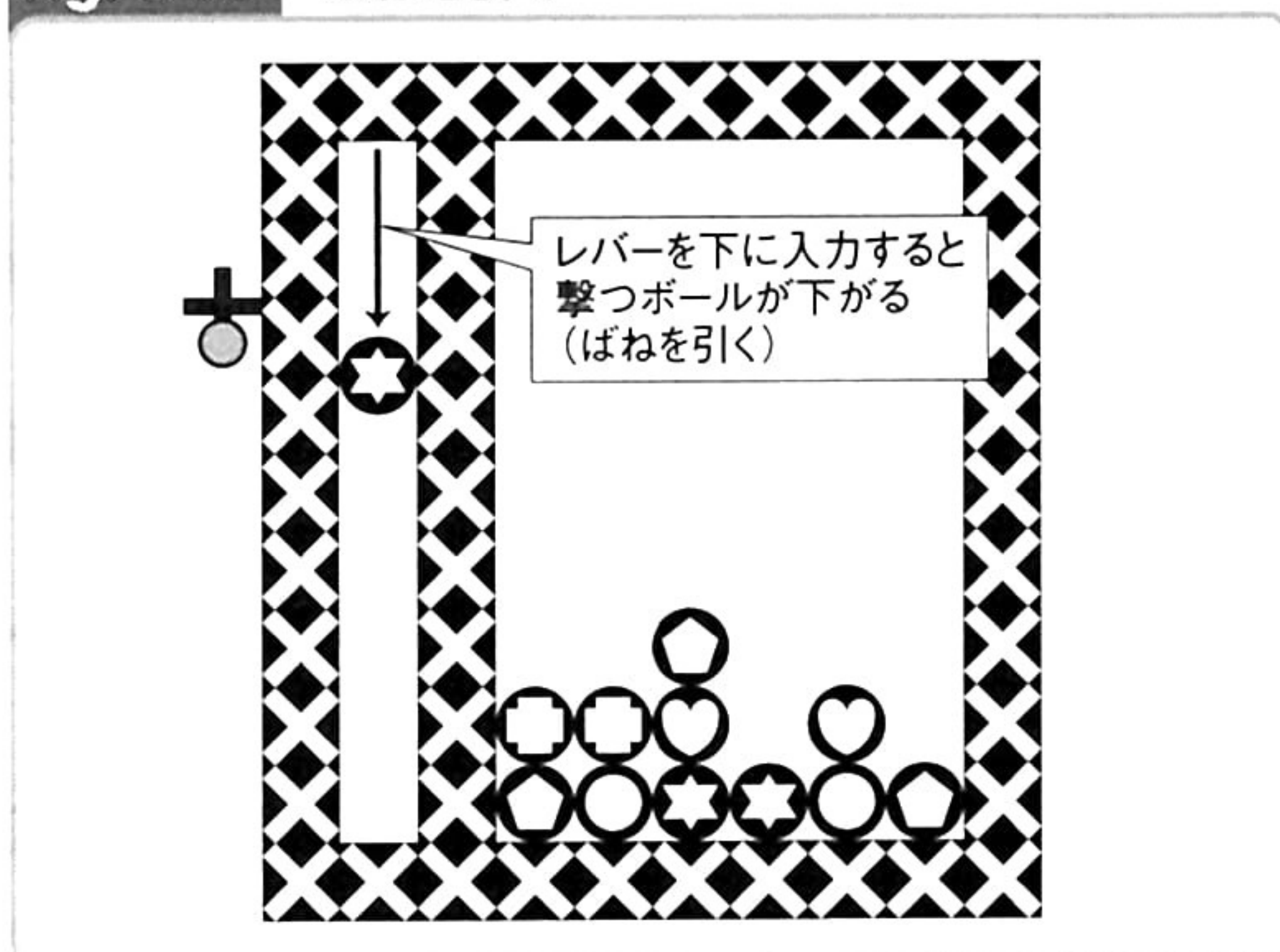



Fig. 5-78 ボールを撃つ

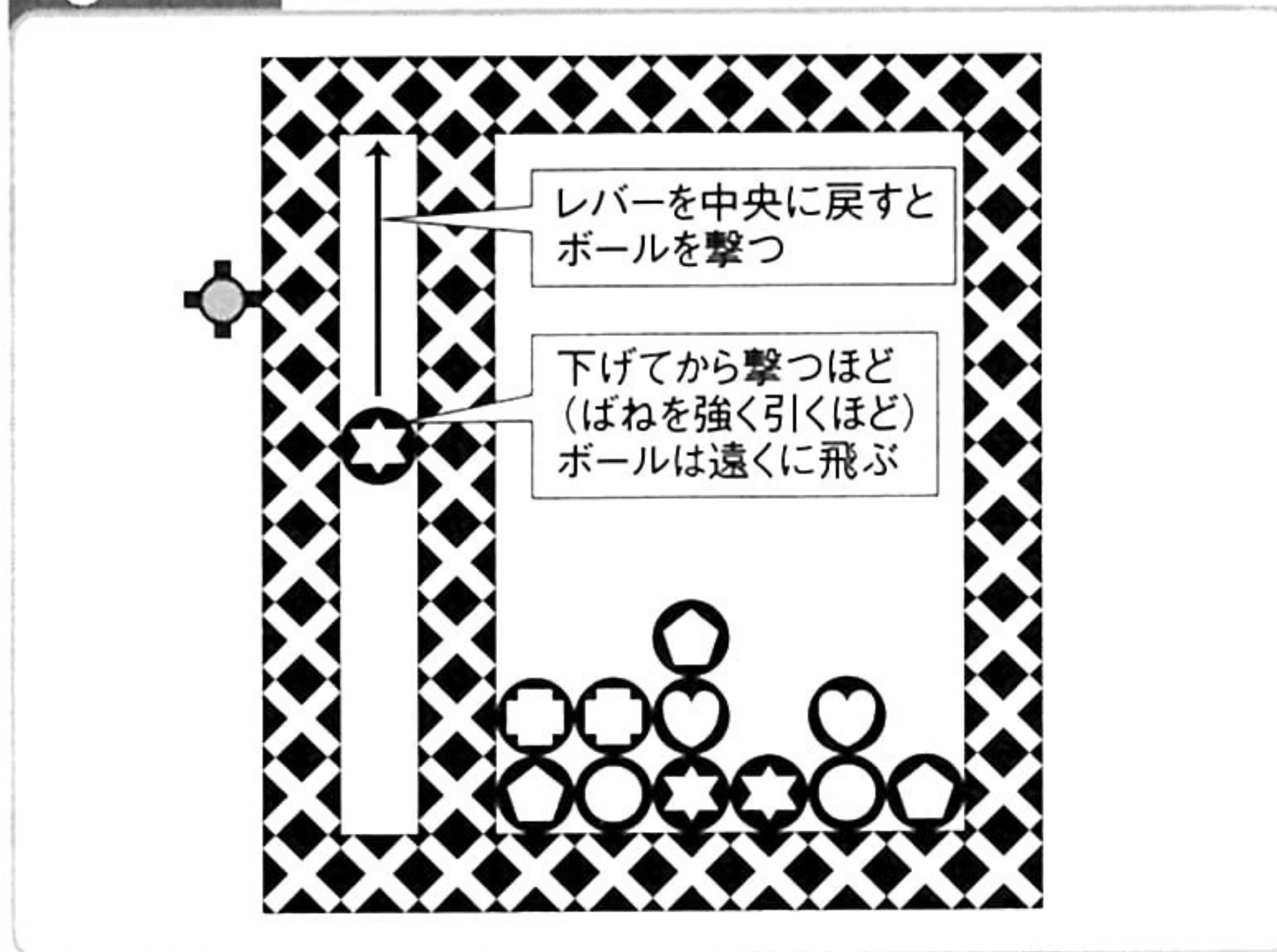



Fig. 5-79 撃ったボールが落ちてくる

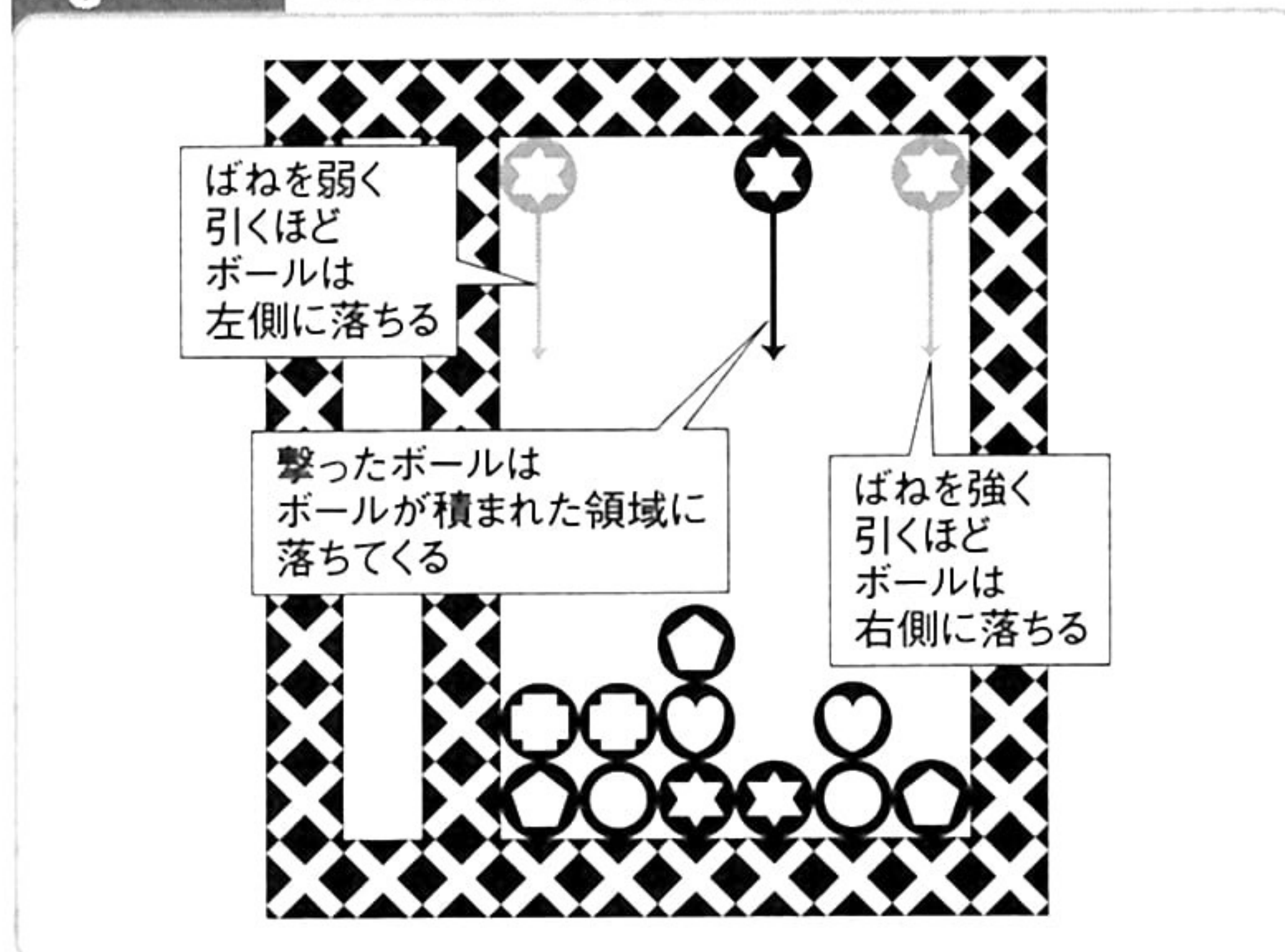
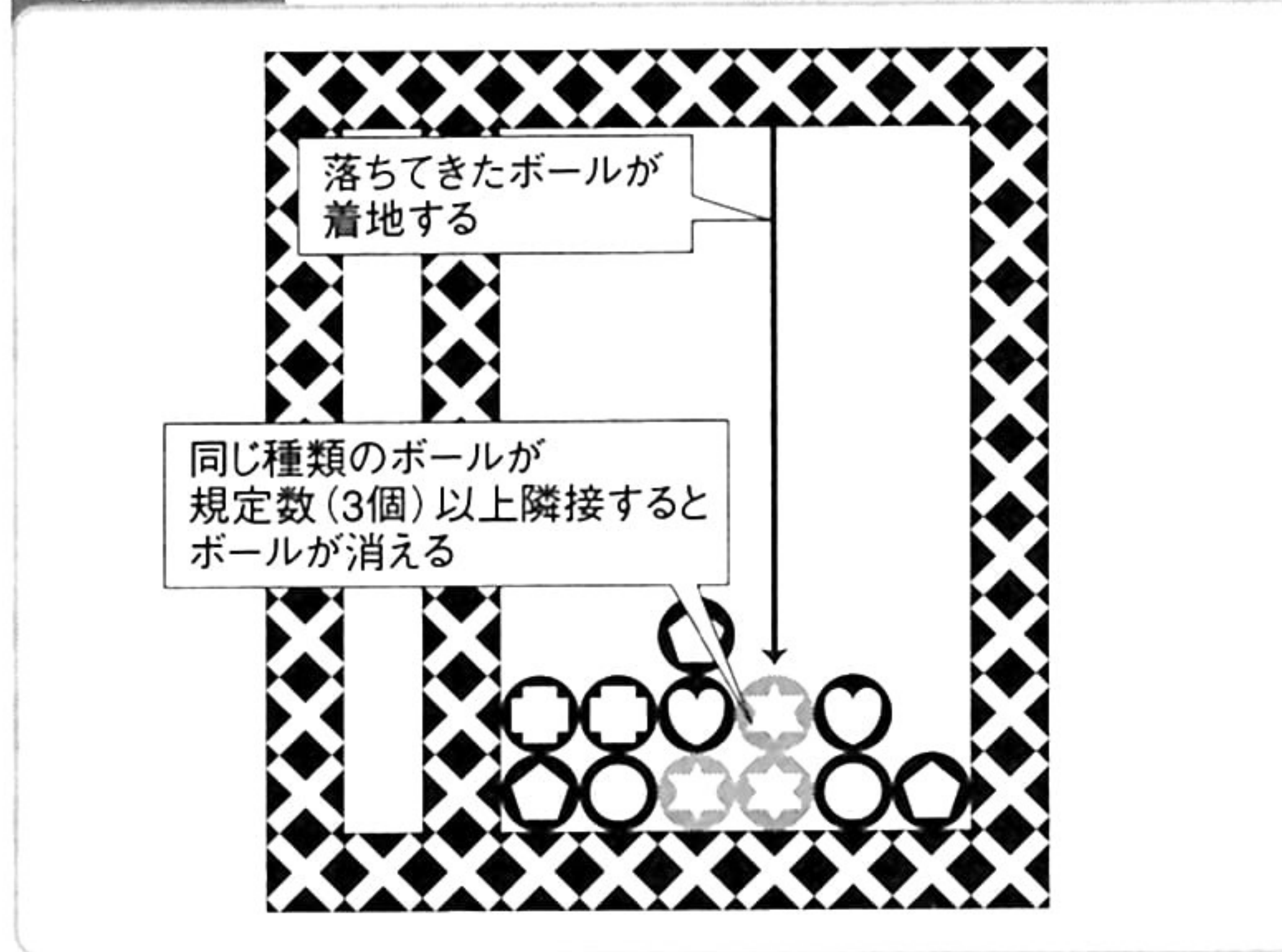



Fig. 5-80 ボールが消える

アルゴリズム

ばねの力でボールを弾くには、レバーを下に入力したときに、ばねの強さを増加させます（Fig. 5-81）。レバーを下に入れているかぎり、少しずつばねの強さを増やし続けます。レバーを中央に戻したら、ボールを撃ちます（Fig. 5-82）。撃った瞬間のばねの強さに応じて、ボールの落下位置を決めます。ばねを強く引くほど、ボールが右側（遠く）に落ちるようにします。

ステージはセルで表現します（Fig. 5-83）。ステージの壁は「＝」、ボールは「0〜4」の数字で表しました。

ボールを撃ったら、ボールの下が空のセルであるかぎり、撃ったボールを落とします。ボールが他のボールや床に着地したら、その場所にボールを固定します（Fig. 5-84）。

ボールが着地したら、同じ種類のボールが規定数（3個）以上隣接しているかどうかを調べます。これは「ボールを入れ替える」（→p. 331）とまったく同じ処理です。すべてのボールについて、上下左右のボールを再帰的に調べ、隣接する同じ種類のボールを探し出します。

Fig. 5-81 ばねの強さを増やす

レバーを下に入力したら
ばねの強さを増加させる

＜ばねの強さ＞
2.5

レバーを下に入れているかぎり
ばねの強さを増やし続ける

＜ばねの強さ＞
4.0

Fig. 5-82 ボールの落下位置を決める

ばねが最弱のときの
落下位置（左端）

ばねが最強のときの
落下位置（右端）

ばねの強さに比例して
ボールの落下位置を
右側にする

Fig. 5-83 ステージをセルで表現する

壁
（＝）

ボール
（0〜4）

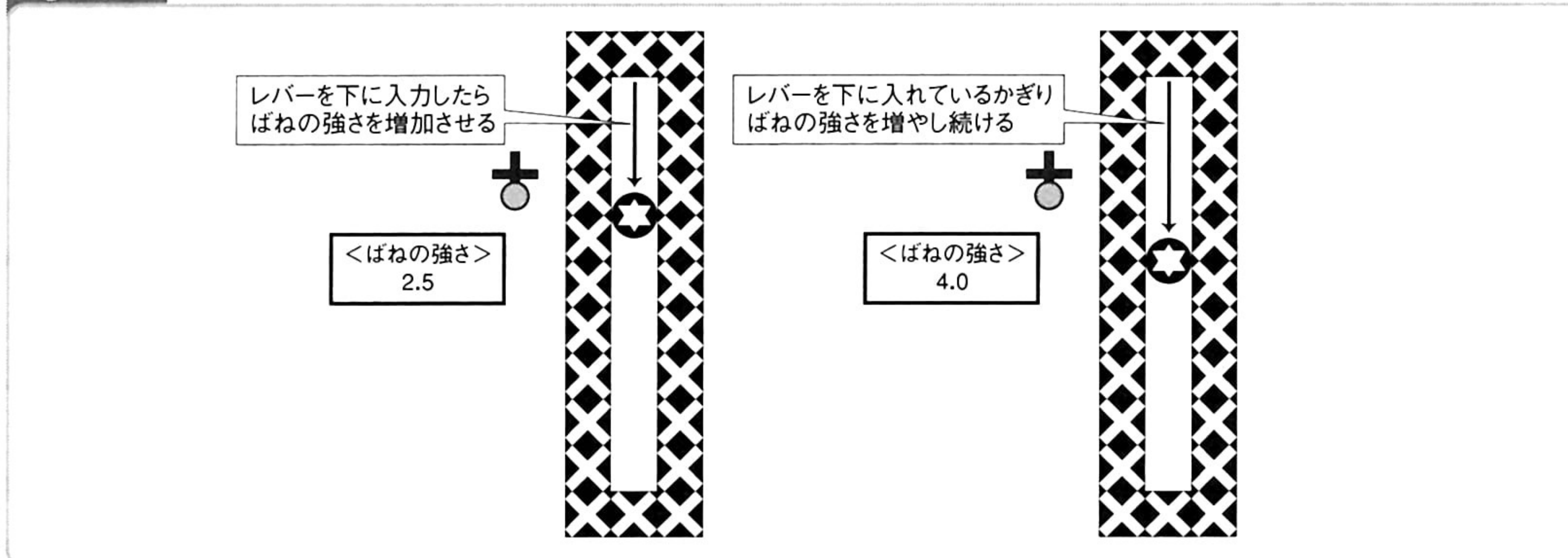
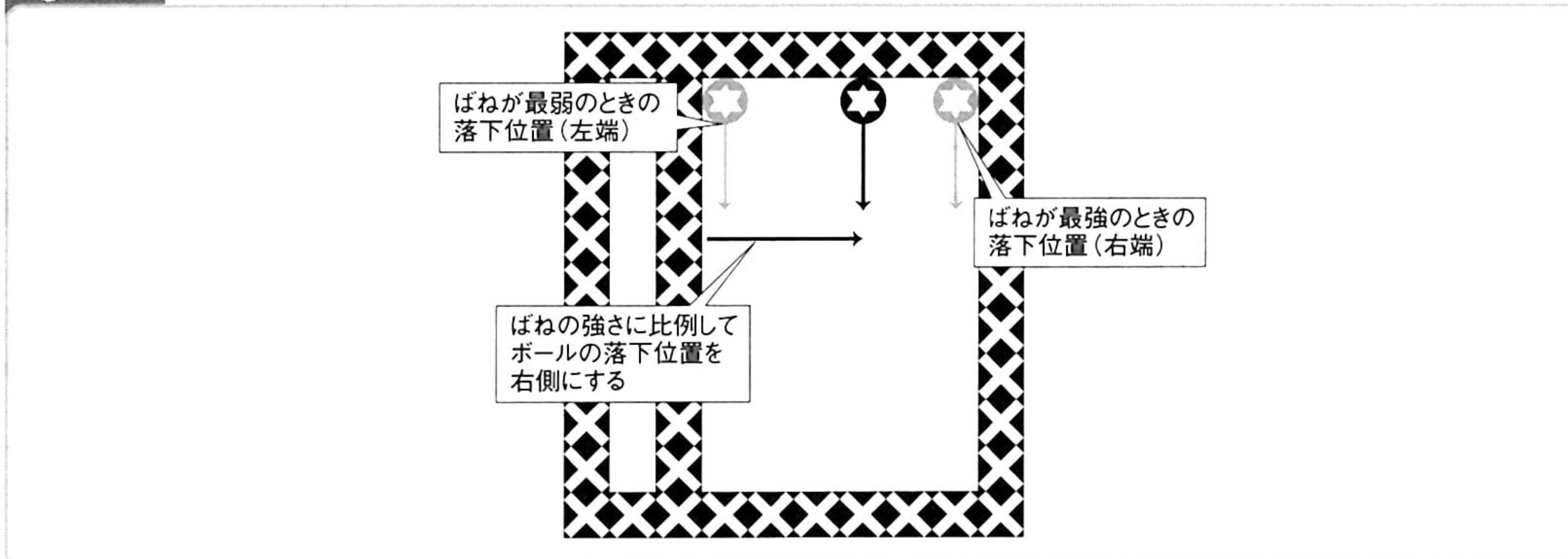
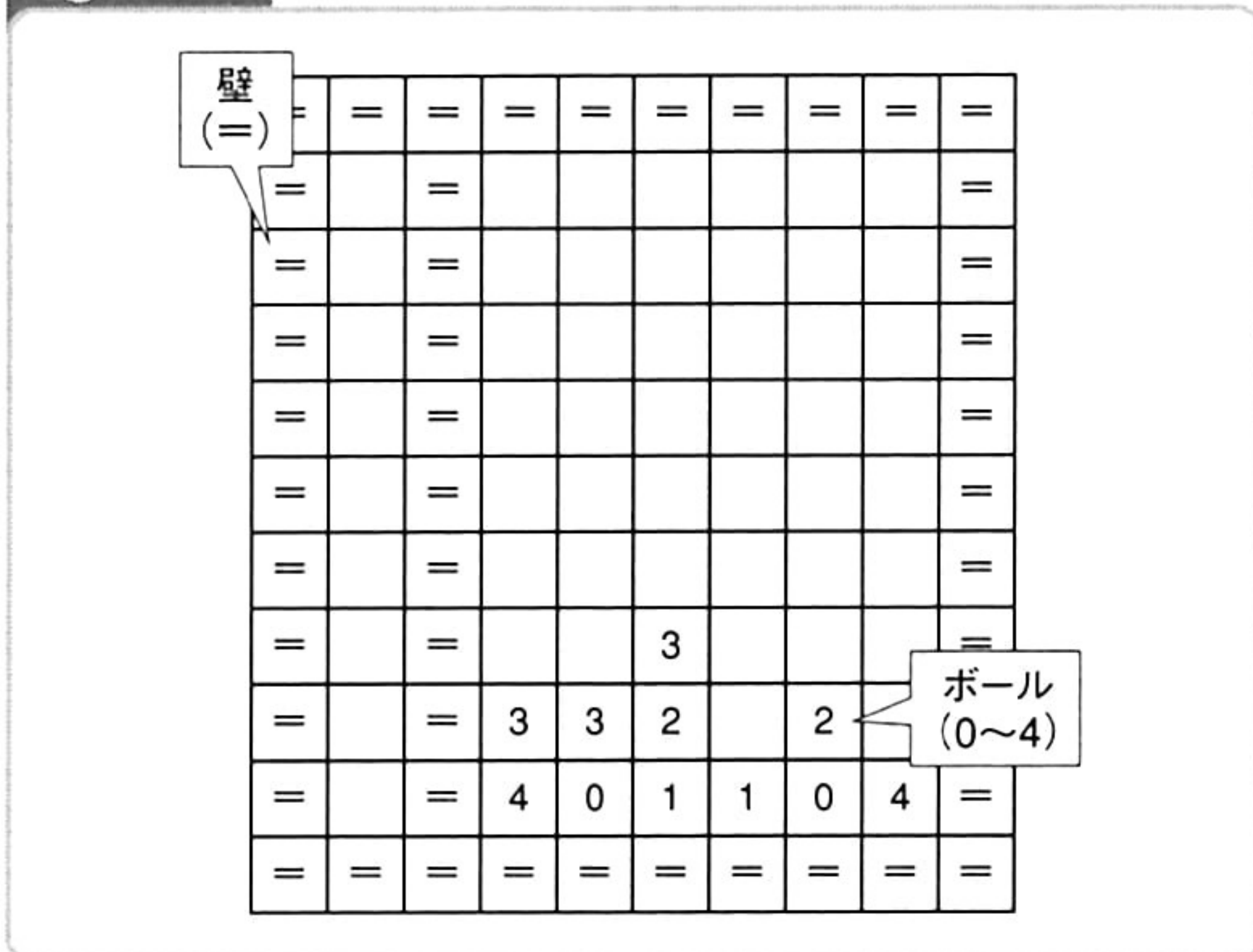
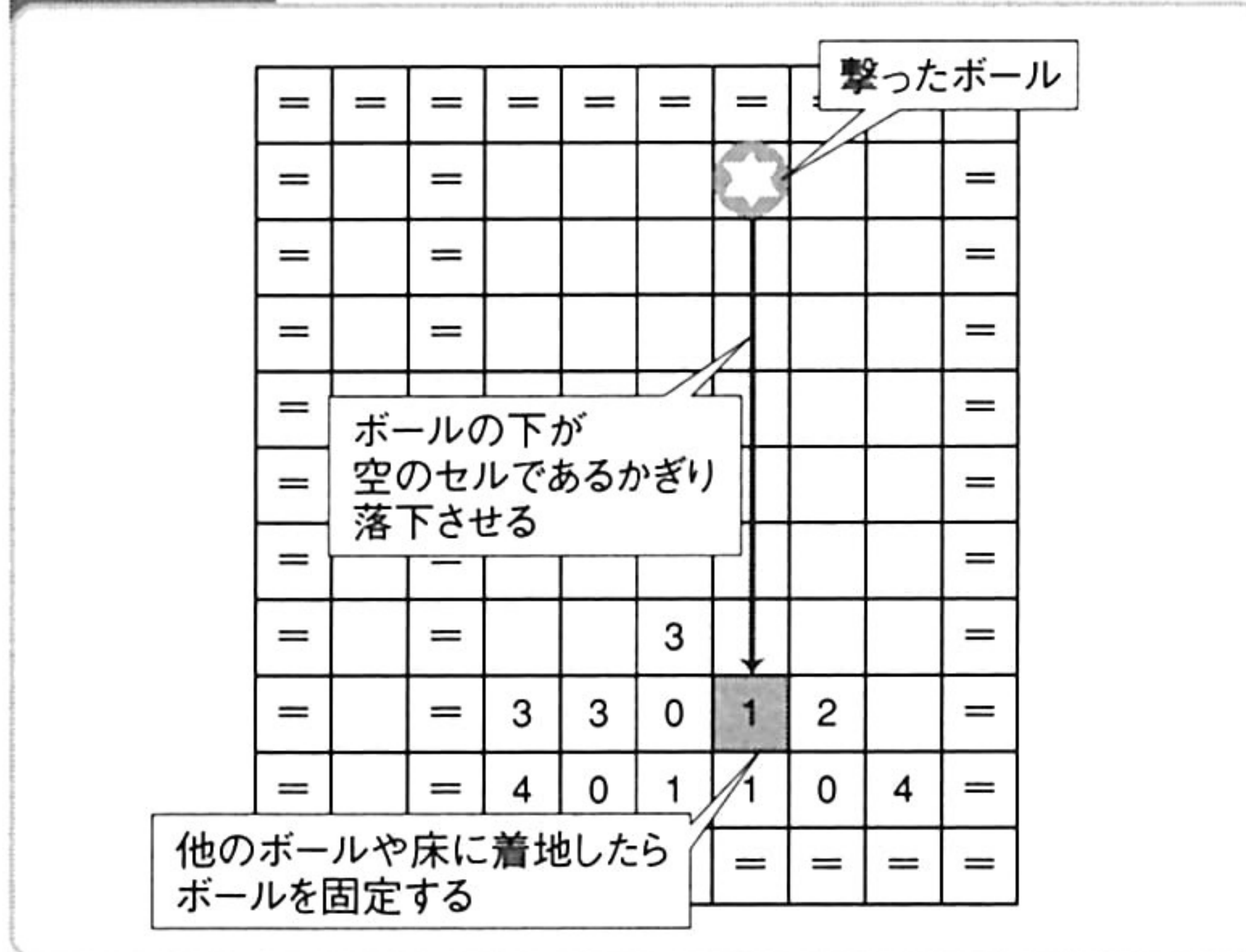
＝	＝	＝	＝	＝	＝	＝	＝	＝	＝
＝		＝							＝
＝		＝							＝
＝		＝							＝
＝		＝							＝
＝		＝							＝
＝		＝							＝
＝		＝			3				＝
＝		＝	3	3	2		2		＝
＝		＝	4	0	1	1	0	4	＝
＝	＝	＝	＝	＝	＝	＝	＝	＝	＝

Fig. 5-84 ボールを固定する

撃ったボール

ボールの下が
空のセルであるかぎり
落下させる

他のボールや床に着地したら
ボールを固定する

＝	＝	＝	＝	＝	＝	＝			
＝		＝							＝
＝		＝							＝
＝		＝							＝
＝									＝
＝									＝
＝									＝
＝		＝			3				＝
＝		＝	3	3	0	1	2		＝
＝		＝	4	0	1	1	0	4	＝
						＝	＝	＝	＝

プログラム

List 5-14はばねでボールを撃つプログラムです。ステージの移動処理、セルがボールかどうかを判定する処理、隣接したボールを消す処理を掲載しました。

移動処理(Move関数)は、初期状態・入力状態・射出状態・消去判定状態・消去状態に分かれています。初期状態では、ばねを引く力を初期化し、撃つボールの座標や種類を設定します。

入力状態では、レバーを下に入力したら、ばねを引く強さを増やします。レバーを下に入れるのをやめたら、ボールを撃つために、射出状態に移行します。

射出状態では、ボールを上に移動して、撃ち出される様子を表現します。ボールがステージの上端に達したら、ステージの右側の領域に落下させます。ボールを落とす位置は、ばねを強く引くほど右にします。

ボールを落としたら、落下状態に移行します。ここではボールの下のセルを調べて、セルが空であるかぎり、ボールを落とします。ボールが着地したら、ボールの種類をセルに書き込んでから、消去判定状態に移行します。

消去判定状態では、同じ種類のボールが規定数(3個)以上並んでいるかどうかを調べて、並んでいたらボールを消します。ボールを消えるボールに変化させ、消去状態に移行します。消去状態では、一定時間が経過するのを待ってから、消えるボールを完全に削除します。

セルがボールかどうかを判定する処理(IsBall関数)は、セルがボールのときにtrueを返します。隣接したボールを消す処理(Erase関数)は、「ボールを入れ替える」(→p. 331)と共通のプログラムで、隣接した同じ種類のボールを数える処理と、ボールを消す処理の両方に使います。

List 5-14 ばねでボールを撃つ(CFlippedBallStageクラス)

```
// 移動処理
bool CFlippedBallStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 初期状態
    if (State==0) {

        // ばねを引く力を設定する
        Power=0;

        // ボールの座標と種類を設定する
        BX=1;
        BallType='0'+Rand.Int31()%FLIPPED_BALL_TYPE;

        // 入力状態に移行する
        State=1;
    }
```

```
//入力状態
if (State==1) {

    //レバーを下に入力したら、ばねを引く力を強くする
    if (is->Down && Power<FLIPPED_BALL_POWER) {
        Power+=0.1f;
    }

    //レバーを下に入れていなかったら、射出状態に移行する
    if (!is->Down && Power>0) {
        State=2;
    }

    //ばねを引く力に応じて、ボールの座標を変化させる
    BY=2+Power*(10-2)/FLIPPED_BALL_POWER;
}

//射出状態
if (State==2) {

    //ボールがステージの上端に達するまで、上に移動する
    if (BY>2) {
        BY-=0.5f;
    } else

    //ステージの上端に達したら、
    //ステージの右側の領域に移動する
    {
        //ボールのX座標は、ばねの引く強さに応じて設定する
        BX=3+(int)Power;

        //ボールのY座標は、ステージの上端に設定する
        BY=2;

        //落下状態に移行する
        State=3;
    }
}

//落下状態
if (State==3) {

    //ボールの下のセルが空ならば、ボールを落下させる
    if (Cell->Get(BX, (int)BY+1)==' ') {
        BY+=0.5f;
    } else

    //ボールの下のセルが空でなければ、
    //ボールをその場所に固定する
    {
```

```
            // ボールの種類をセルに書き込む
            Cell->Set(BX, (int)BY, BallType);

            // 消去判定状態に移行する
            State=4;
        }
    }

    // 消去判定状態
    if (State==4) {

        // ボールが消えなかったら、初期状態に移行する
        State=0;

        // すべてのセルについて、
        // 同じ種類のボールが規定数(3個)以上
        // 隣接しているかどうかを調べる
        for (int y=0; y<ys; y++) {
            for (int x=0; x<xs; x++) {

                // 同じ種類のボールが規定数以上隣接していたら、
                // ボールを消す
                char c=Cell->Get(x, y);
                if (
                    IsBall(c) &&
                    Erase(x, y, c, 0x80)>=FLIPPED_BALL_ERASE
                ) {
                    // ボールを消えるボールに変化させる
                    Erase(x, y, c|0x80, 0x40);

                    // タイマーを設定して、
                    // 消去状態に移行する
                    Time=0;
                    State=5;
                }
            }
        }

        // ボールのカウントずみのマークを解除する
        for (int y=0; y<ys; y++) {
            for (int x=0; x<xs; x++) {
                Cell->Set(x, y, Cell->Get(x, y)&0x7f);
            }
        }
    }

    // 消去状態
    if (State==5) {

        // 一定時間が経過するのを待ってから、
```

```
            //消えるボールを完全に消去する
            Time++;
            if (Time==30) {
                for (int x=0; x<xs; x++) {
                    for (int y=0; y<ys; y++) {

                        //消えるボールを見つけたときの処理
                        if (Cell->Get(x, y)&0x40) {

                            //消えるボールの上にあるボールを
                            //1段ずつ落下させる
                            for (int i=y; i>2; i--) {
                                Cell->Set(x, i, Cell->Get(x, i-1));
                            }

                            //ステージの上端は空のセルにする
                            Cell->Set(x, 2, ' ');
                        }
                    }
                }

                //消去判定状態に移行する
                State=4;
            }
        }

        return true;
    }

    //セルがボールかどうかを判定する処理
    bool CFlippedBallStage::IsBall(char c) {
        return '0'<=c && c<'0'+SWAPPED_BALL_TYPE;
    }
```

SAMPLE

「FLIPPED BALL」は「ばねでボールを撃つ」のサンプルです。

レバーの下（カーソルキーの下）で、ステージ左上のボールが下がります。レバーを下に入れ続けると、ステージの下端に達するまで、ボールは少しずつ下がり続けます。

レバーを中央に戻す（カーソルキーを離す）と、ボールを撃ちます。ボールを大きく下げておくほど、ボールは遠くまで飛びます。

撃ったボールはステージ右側の領域に落ちてきます。ボールが着地したときに、同じ種類のボールが規定数（3個）以上隣接すると、ボールが消えます。

FLIPPED BALL → p. 389

転がる大量のボール

大量のボールがステージ内を転がるアクションです。左右に転がったり、穴から落ちたり、他のボールとぶつかったりしながら、ボールはステージ下方へと落ちていきます。パチンコのような動きが楽しめます。

本書のサンプルでは、ステージ上方にボールが出現します(Fig. 5-85)。ボタン0(Zキー)を押すと左上に、ボタン1(Xキー)を押すと右上に、ボールが発生します。

ボールは左または右に転がっていきます(Fig. 5-86)。左右に他のボールや壁があると、進行方向を変えます。下に何もないと、ボールは落下します。

転がる大量のボールは『ロットロット』に採用されています。このゲームでは、転がる大量のボールをゴールまで導きます。「遅れて追随するカーソル」(→p. 43)で解説したように、カーソルを操作して、ボールの位置を入れ替えることができます。上手にボールを入れ替えることによって、より高いスコアが得られるゴールにボールを導くことが、ゲームの目的です。

Fig. 5-85 ステージの構成

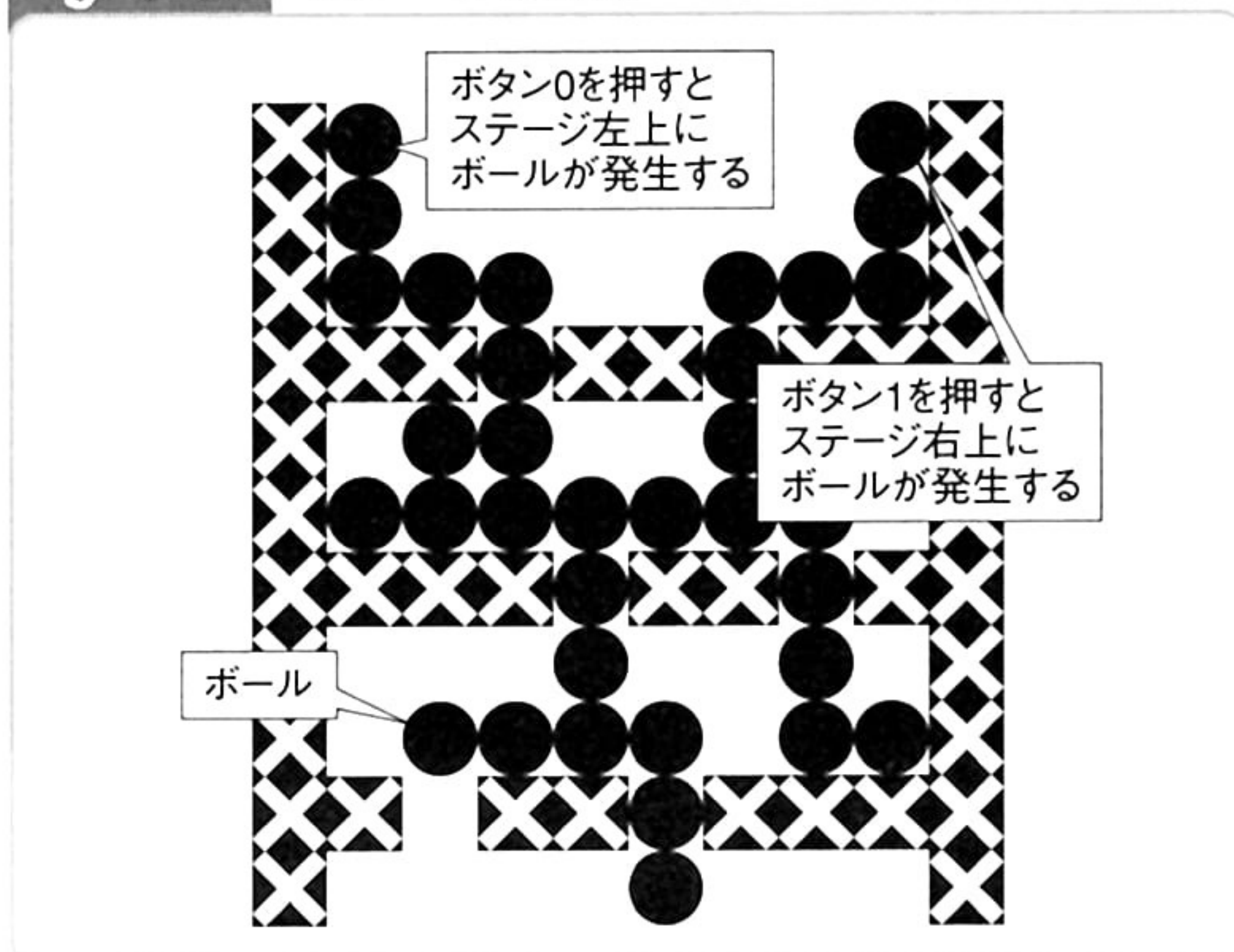
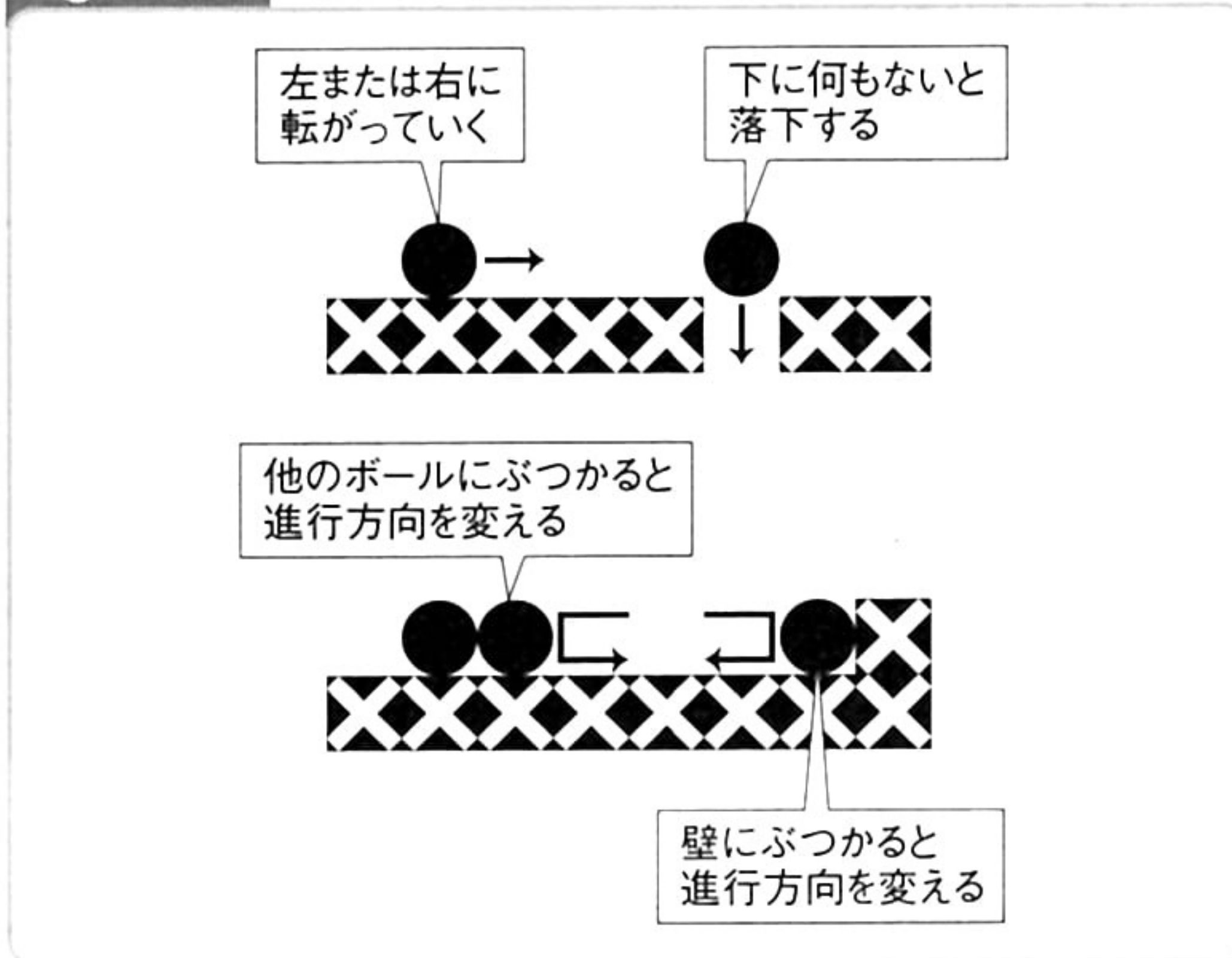



Fig. 5-86 ボールの動き

アルゴリズム

転がる大量のボールを実現するには、各ボールの座標を個別に管理する方法もありますが、ここではセルを使った方法を紹介します(Fig. 5-87)。ステージの壁を「=」、左向きのボールを「0」、右向きのボールを「1」で表します。

ボールを動かすには、ステージ内のすべてのボールを順番に処理します。左向きのボールについては、ボールの「下・左・右」のセルを調べて、セルが空ならば、その方向にボールを進

めます (Fig. 5-88)。右のセルが空のときには、ボールの進行方向を右に変えて、ボールを右に進めます。

右向きのボールについても、左向きのボールと同様に動かします (Fig. 5-89)。ボールの「下・右・左」のセルを調べて、セルが空ならば、その方向にボールを進めます。左のセルが空のときには、ボールの進行方向を左に変えて、ボールを左に進めます。

セルを使うと、大量のボールを比較的軽い処理で動かすことができます。本書のサンプルでは、ボールをセル単位で動かしていますが、セルが十分細かければ、ボールは滑らかに動いているように見えます。

一方、「自律的に動くキャラクター」(→p. 36) のように、各ボールの座標を個別に管理して動かす方法もあります。この場合、ボールを動かす処理は少し重くなりますが、より複雑な動きが可能になります。動かすボールの個数や、動きの複雑さに応じて、適切な方法を選ぶとよいでしょう。

Fig. 5-87　ステージをセルで表現する

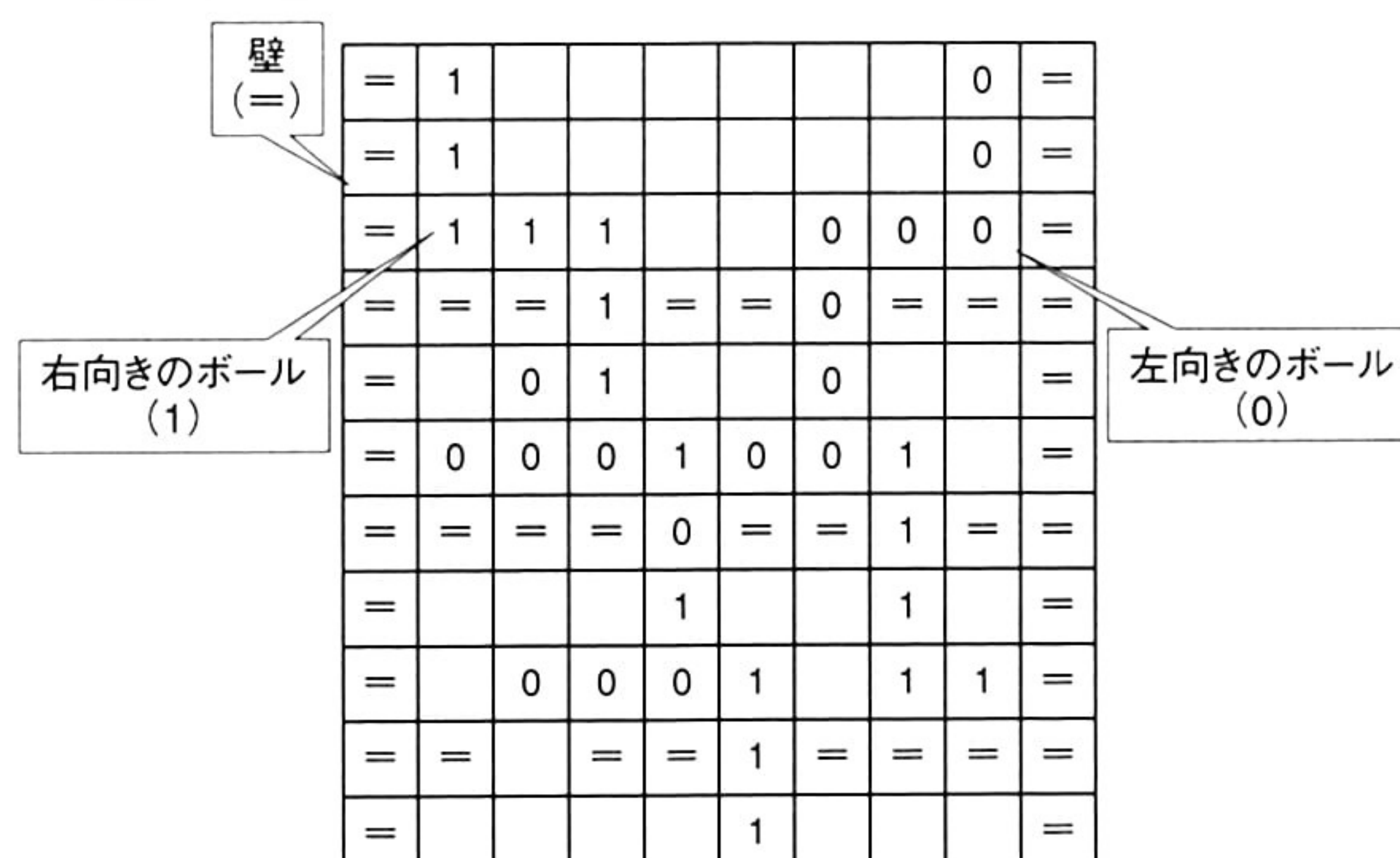
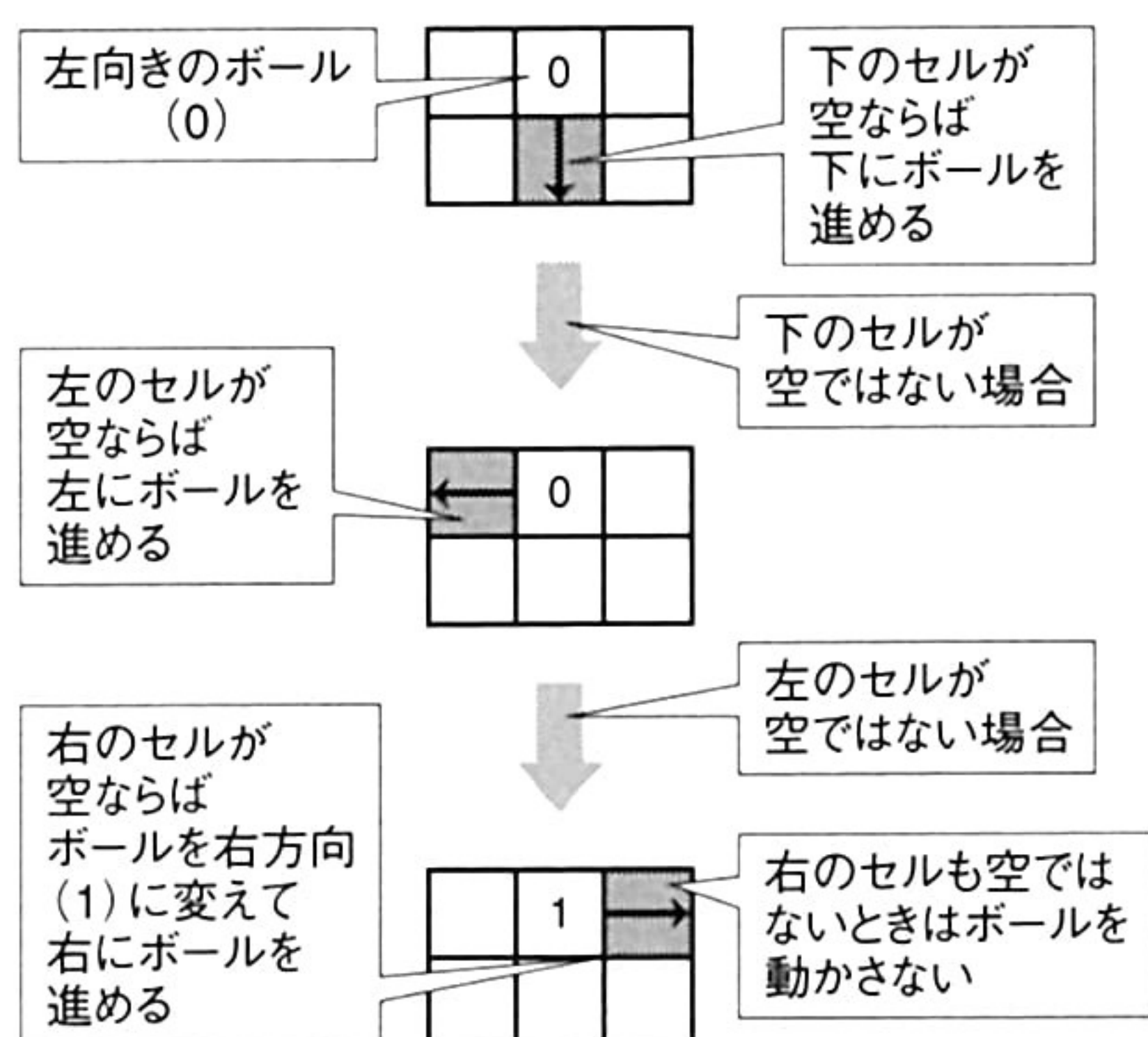
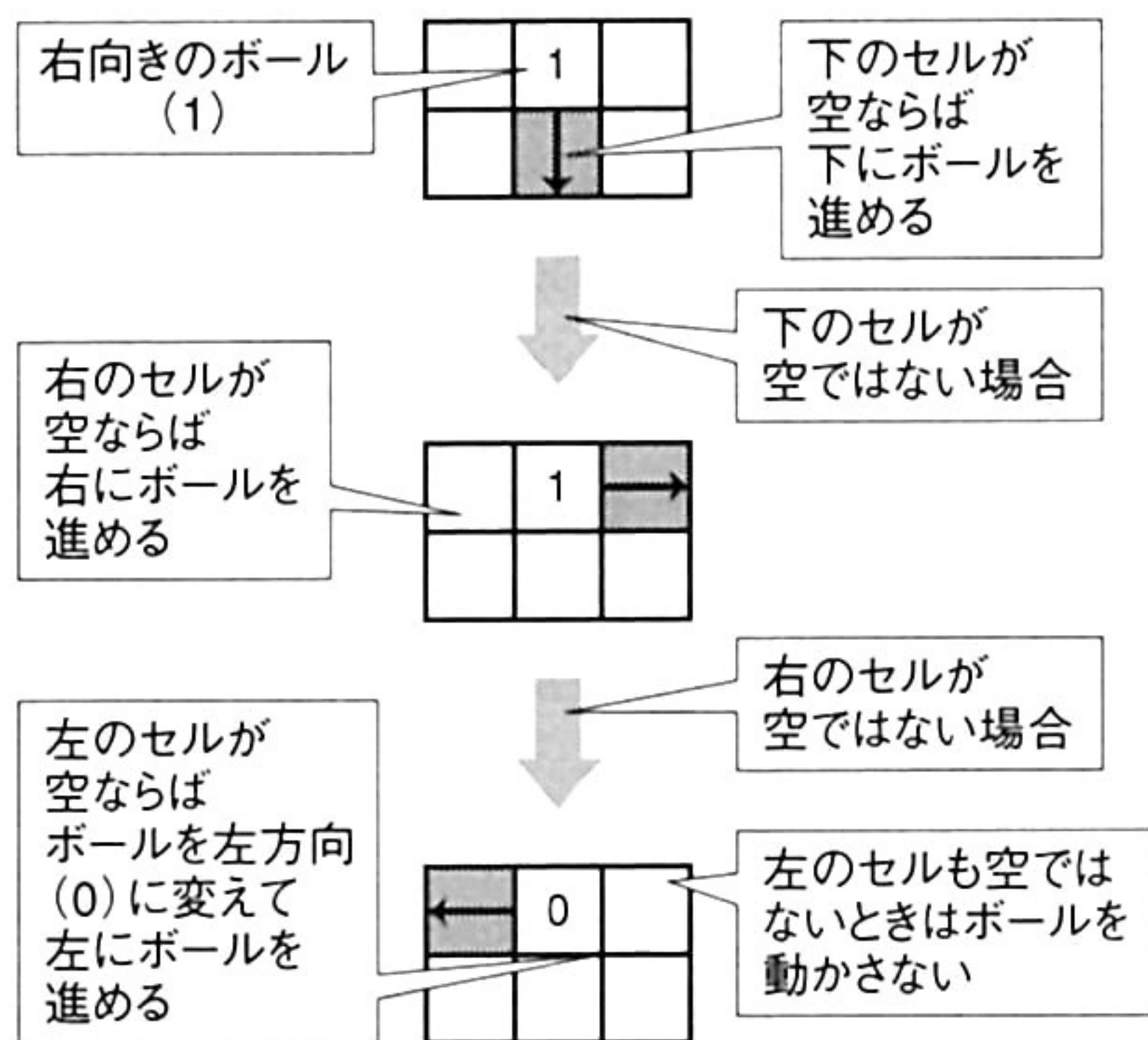
壁 (＝)
右向きのボール (1)
左向きのボール (0)

＝	1							0	＝
＝	1							0	＝
＝	1	1	1			0	0	0	＝
＝	＝	＝	1	＝	＝	0	＝	＝	＝
＝		0	1			0			＝
＝	0	0	0	1	0	0	1		＝
＝	＝	＝	＝	0	＝	＝	1	＝	＝
＝				1			1		＝
＝		0	0	0	1		1	1	＝
＝	＝		＝	＝	1	＝	＝	＝	＝
＝					1				＝

Fig. 5-88　左向きのボールを動かす

左向きのボール (0)
下のセルが空ならば下にボールを進める
下のセルが空ではない場合
左のセルが空ならば左にボールを進める
左のセルが空ではない場合
右のセルが空ならばボールを右方向 (1) に変えて右にボールを進める
右のセルも空ではないときはボールを動かさない

Fig. 5-89　右向きのボールを動かす

右向きのボール (1)
下のセルが空ならば下にボールを進める
下のセルが空ではない場合
右のセルが空ならば右にボールを進める
右のセルが空ではない場合
左のセルが空ならばボールを左方向 (0) に変えて左にボールを進める
左のセルも空ではないときはボールを動かさない

プログラム

List 5-15は転がる大量のボールのプログラムです。ステージの移動処理を掲載しました。

移動処理では、ボタンの入力に応じて、ステージの上方に新しいボールを出現させます。ボタン0を押したときには左上に、ボタン1を押したときには右上に、ボールを生成します。

次に、ステージ内のセルを調べて、すべてのボールを動かします。左向きのボールについては、下・左・右の順に隣接するセルを調べて、セルが空の方向にボールを動かします。右向きのボールについては、下・右・左の順にセルを調べて、空の方向にボールを動かします。

左向きのボールを右に進めるときには、進行方向を右に変えます。右向きのボールを左に進めるときには、進行方向を左に変えます。一度ボールの進行方向を変えたら、他のボールに衝突するまで、同じ方向に動かし続けます。

動かしたボールは、最上位ビットを「1」に設定することによって、移動ずみのマークを付けます。これは同じボールを重複して動かさないためです。移動処理を1回行うごとに、各ボールを1回ずつ動かします。

List 5-15 転がる大量のボール(CRollingBallStageクラス)

```
// 移動処理
bool CRollingBallStage::Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // ボタン0を押したときには、
    // ステージの左上に新しいボール(右向き)を出現させる
    if (is->Button[0] && Cell->Get(1, 1)==' ') {
        Cell->Set(1, ys/MAX_Y, '1');
    }

    // ボタン1を押したときには、
    // ステージの右上に新しいボール(左向き)を出現させる
    if (is->Button[1] && Cell->Get(xs-2, 1)==' ') {
        Cell->Set(xs-2, ys/MAX_Y, '0');
    }

    // ステージ内のすべてのボールを動かす
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {

            // セル、
            // 移動ずみのボールのセル、
            // 進行方向を逆にした移動ずみのボールのセル
            char
                c=Cell->Get(x, y),
```

```
    d=c|0x80,
    e=('0'+'1'-c)|0x80;

// 左向きのボールを動かす
if (c=='0') {

    // 現在位置のセルを空にする
    Cell->Set(x, y, ' ');

    // ステージの下端に達していなければ、ボールを動かす
    if (y<ys-1) {

        // 下が空のセルならば、ボールを落下させる
        if (Cell->Get(x, y+1)==' ') {
            Cell->Set(x, y+1, d);
        } else

        // 左が空のセルならば、ボールを左に進める
        if (Cell->Get(x-1, y)==' ') {
            Cell->Set(x-1, y, d);
        } else

        // 右が空のセルならば、進行方向を反転して、ボールを右に進める
        if (Cell->Get(x+1, y)==' ') {
            Cell->Set(x+1, y, e);
        } else

        // その他の場合には、ボールを動かさない
        {
            Cell->Set(x, y, d);
        }
    }
} else

// 右向きのボールを動かす
if (c=='1') {

    // 現在位置のセルを空にする
    Cell->Set(x, y, ' ');

    // ステージの下端に達していなければ、ボールを動かす
    if (y<ys-1) {

        // 下が空のセルならば、ボールを落下させる
        if (Cell->Get(x, y+1)==' ') {
            Cell->Set(x, y+1, d);
        } else

        // 右が空のセルならば、ボールを右に進める
        if (Cell->Get(x+1, y)==' ') {
```

```
                    Cell->Set(x+1, y, d);
                } else

                // 左が空のセルならば、進行方向を反転して、ボールを左に進める
                if (Cell->Get(x-1, y)==' ') {
                    Cell->Set(x-1, y, e);
                } else

                // その他の場合には、ボールを動かさない
                {
                    Cell->Set(x, y, d);
                }
            }
        }
    }
}

// ステージ内のすべてのボールについて、移動ずみのマークを解除する
for (int y=0; y<ys; y++) {
    for (int x=0; x<xs; x++) {
        Cell->Set(x, y, Cell->Get(x, y)&0x7f);
    }
}
return true;
}
```

SAMPLE

「ROLLING BALL」は「転がる大量のボール」のサンプルです。

ボタン0（Zキー）押すとステージ左上に、ボタン1（Xキー）を押すとステージ右上に、ボールが出現します。出現したボールは、壁や他のボールの上を左右に転がっていき、下に何もないときには落下します。ステージの下端から出ると、ボールは消滅します。

ボールが他のボールにぶつかると、進行方向を左右に変えます。大量のボールがあると、ボール同士が干渉して、複雑な動きを見せながら下に落ちていきます。

ROLLING BALLL → p. 389

まとめ

本章では「ボール」を使ったさまざまなアクションを紹介しました。ブロックと同様に、ボールを使ったパズルゲームは数多くあります。ブロックとボールのどちらを使ってもかまわないゲームもありますが、ボールならではの丸い形を生かしたゲームもあります。ボールの特徴を使ったゲームには、ブロックのゲームとはまた違った、繊細で滑らかな動きを楽しめる作品が多く見受けられます。

というわけで、「ボールのゲームを作るときには、ボールならではの形や動きを生かそう！」というのが本章のまとめです。

Stage 06

その他

Others

パズルゲームは非常に数多くの種類がリリースされています。ゲームの内容も千差万別ですが、本書では「動かす」「落とす」「つなぐ」「ブロック」「ボール」といった、多くのゲームに共通する要素に注目して、パズルゲームのいろいろなパターンを紹介してきましたが、本章では、今までの分類に入らなかった、少し変わったアクションを紹介します。

アイテムの位置を記憶する

ステージ上に短時間だけ表示されるアイテムの位置を憶えておいて、そのアイテムを回収するアクションです。回収するときには、アイテムは表示されないので、記憶力だけが頼りです。

ステージの開始時に、短い時間だけアイテムが表示されます（Fig. 6-1）。アイテムはすぐに消えてしまうので、この間にアイテムの位置をできるだけ憶えます。

アイテムが消えたら、カーソルを上下左右に動かして、アイテムの位置でボタンを押します（Fig. 6-2）。うまくカーソルをアイテムに重ねてボタンを押すと、アイテムを回収することができます。回収時にはアイテムが見えないので、記憶だけが頼りです。回収に成功したアイテムは見えるようになります。

アイテムの位置を記憶するゲームには『ドリームショッパー』があります。このゲームはステージ内に多数のアイテムが配置されています。プレイヤーはキャラクターを動かして、アイテムを回収します。アイテムはステージ開始時に短時間だけ表示されますが、回収時には表示されません。

アイテムには得点があり、一定以上の得点を稼ぐと、ステージをクリアすることができます。アイテムの得点はそれぞれ異なるので、アイテムが表示されている間に、高得点のアイテムがどこにあるのかを記憶しておくことが重要です。

Fig. 6-1 アイテムが表示される

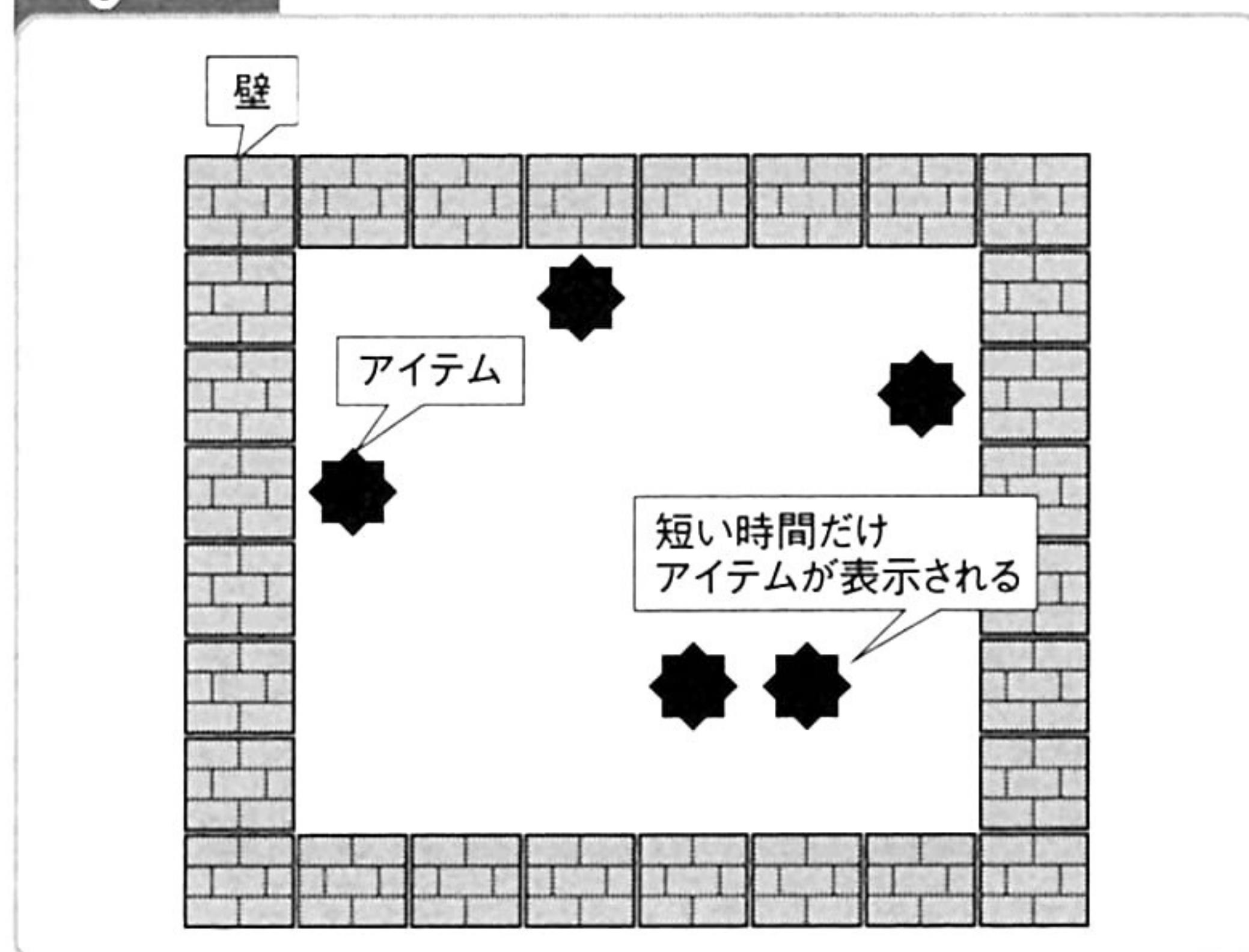
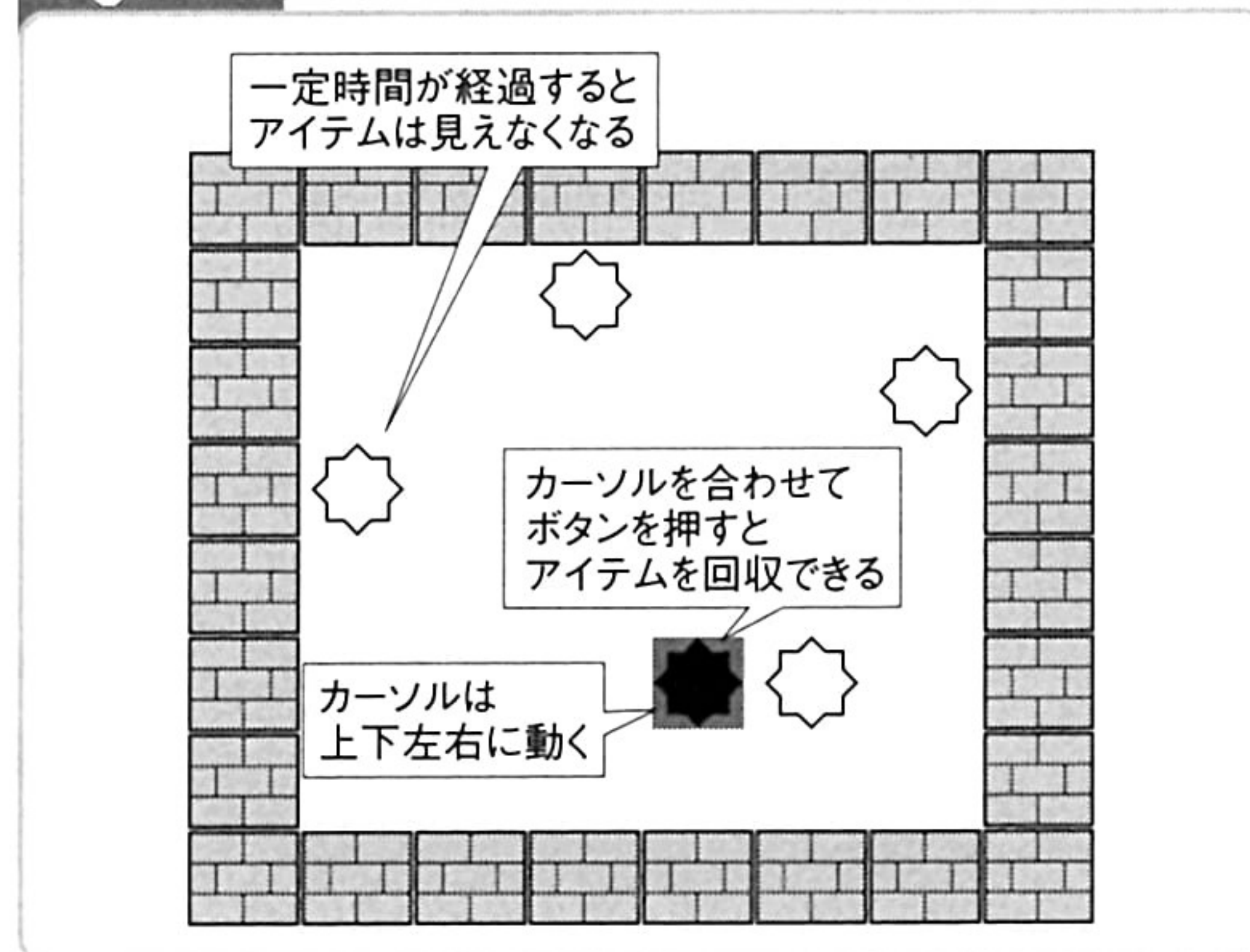



Fig. 6-2 アイテムを回収する

アルゴリズム

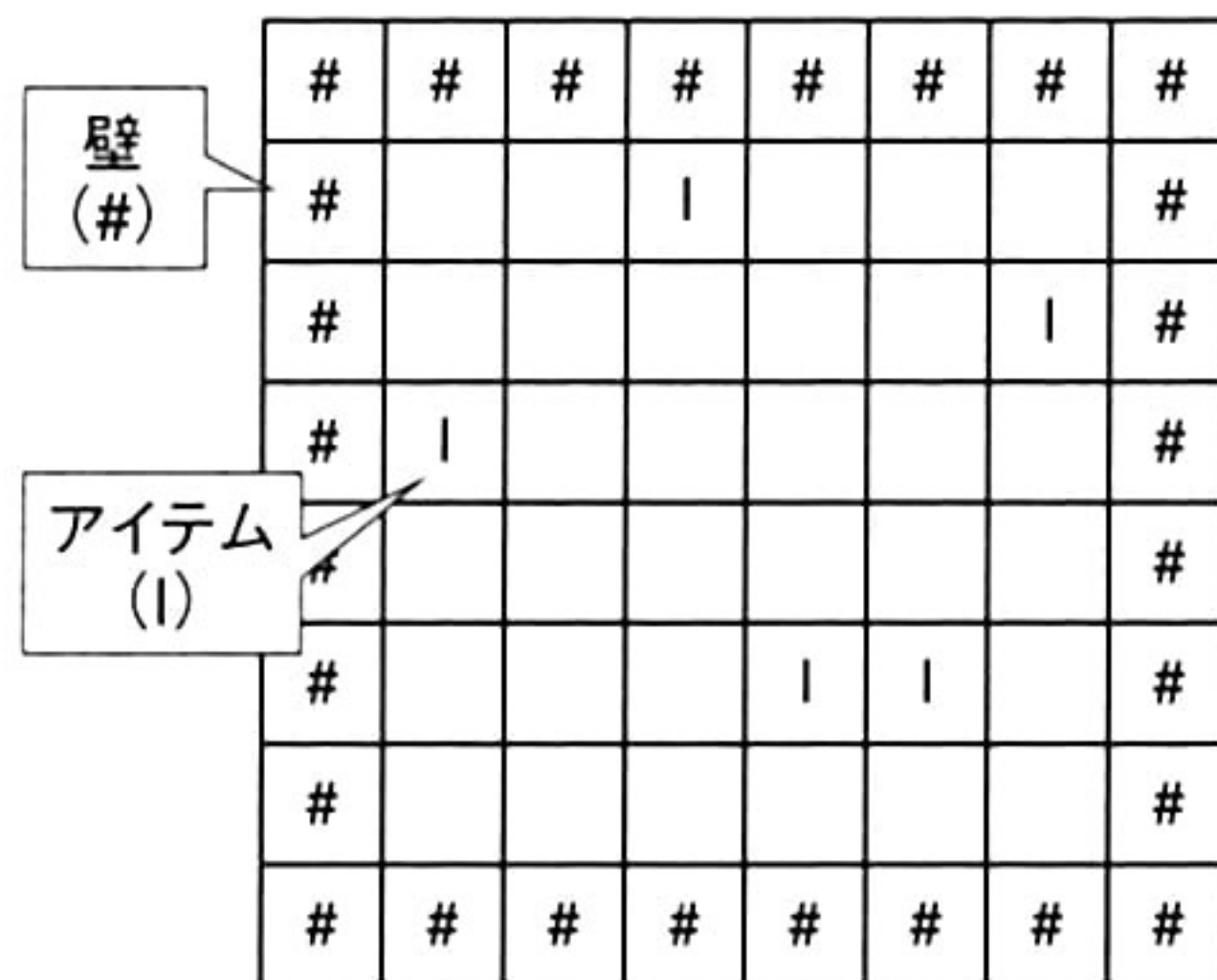
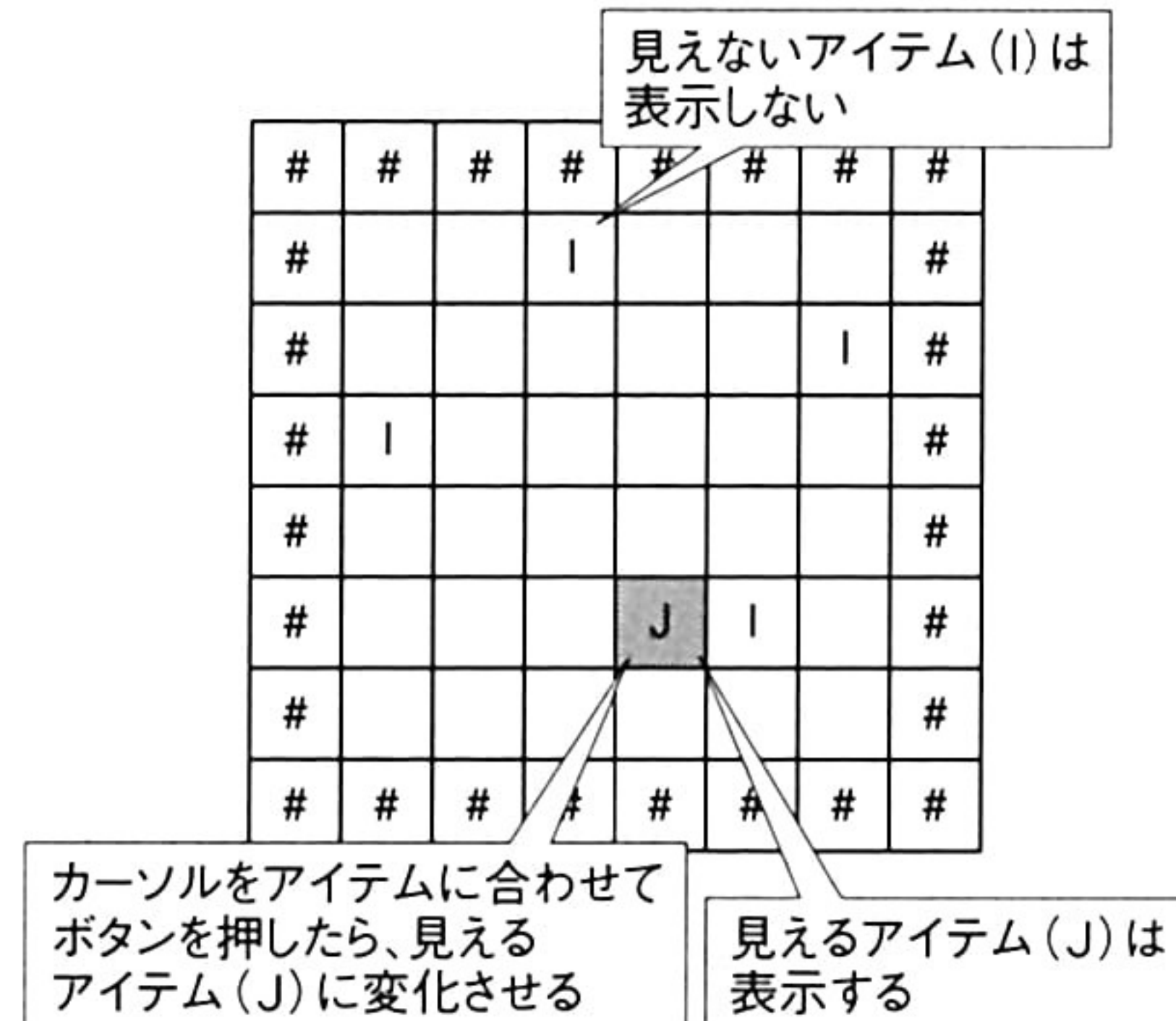
アイテムの位置を記憶するアクションを実現するには、まずステージをセルで表現します（Fig. 6-3）。ステージの壁を「#」、アイテムを「I」で表現しました。

Fig. 6-3 ステージをセルで表現する

#	#	#	#	#	#	#	#
#			I				#
#						I	#
#	I						#
#							#
#				I	I		#
#							#
#	#	#	#	#	#	#	#

壁（#）
アイテム（I）

Fig. 6-4 アイテムを見えるアイテムにする

#	#	#	#	#	#	#	#
#			I				#
#						I	#
#	I						#
#							#
#				J	I		#
#							#
#	#	#	#	#	#	#	#

見えないアイテム（I）は表示しない
カーソルをアイテムに合わせてボタンを押したら、見えるアイテム（J）に変化させる
見えるアイテム（J）は表示する

ステージの開始時は、一定時間だけアイテムの位置を表示します。セルを調べて、アイテム(I)に対応する位置に、アイテムのグラフィックを表示します。ステージが始まったら、アイテムは表示しません。

カーソルをアイテムの位置に合わせて、ボタンを押したときには、アイテム(I)を見えるアイテム(J)に変化させます(Fig. 6-4)。ステージが始まった後は、アイテムは表示しませんが、見えるアイテムは表示します。

プログラム

List 6-1はアイテムの位置を記憶するアクションのプログラムです。ステージの移動処理を掲載しました。

移動処理は表示状態と入力状態に分かれています。表示状態では、一定時間が経過するのを待ってから、入力状態に移行します。表示状態の間に、アイテムの位置を表示します。

入力状態になったら、アイテムは表示しません。レバー入力に応じて、カーソルを上下左右に動かします。カーソルをアイテムに合わせて、ボタン0(Zキー)を押したら、アイテムを見えるアイテムに変化させます。入力状態でも、見えるアイテムは表示します。

ボタン1(Xキー)を押したら、表示状態に移行して、もう一度アイテムの位置を表示します。実際のゲームでは、アイテムの位置を表示する回数を制限するとか、特定の条件を満たしたときだけ表示するといった工夫をするとよいでしょう。

List 6-1 アイテムの位置を記憶する(CMemorizingItemPositionStageクラス)

```
// 移動処理
bool CMemorizingItemPositionStage::Move(const CInputState* is) {

	// 表示状態
```

```
//アイテムの位置を一定時間だけ表示する
if (State==0) {

    //一定時間が経過するのを待ち、入力状態に移行する
    Time++;
    if (Time==60) {
        State=1;
    }
}

//入力状態
if (State==1) {

    //レバー入力に応じて、カーソルを上下左右に動かす
    if (!PrevLever) {
        int cx=CX, cy=CY;
        if (is->Left) cx--;
        if (is->Right) cx++;
        if (is->Up) cy--;
        if (is->Down) cy++;

        //カーソルの移動先が壁でないときには、
        //カーソルを動かす
        if (Cell->Get(cx, cy)!='#') {
            CX=cx;
            CY=cy;
        }
    }
    PrevLever=is->Left||is->Right||is->Up||is->Down;

    //ボタンを押したときの処理
    if (!PrevButton) {

        //アイテムの上でボタン0を押したら、
        //アイテムを見えるアイテムにする
        if (is->Button[0] && Cell->Get(CX, CY)=='I') {
            Cell->Set(CX, CY, 'J');
        } else

        //ボタン1を押したら、
        //アイテムの位置を一定時間だけ表示する
        if (is->Button[1]) {

            //タイマーを設定し、表示状態に移行する
            Time=0;
            State=0;
        }
    }
    PrevButton=is->Button[0]||is->Button[1];
```

```
    }

    return true;
  }
```

SAMPLE

「MEMORIZING ITEM POSITION」は「アイテムの位置を記憶する」のサンプルです。

ステージの開始時に、短時間だけアイテムが表示されます。ここでアイテムの位置をできるだけ憶えます。なお、アイテムが消えた後にボタン1（Xキー）を押すと、再びアイテムの位置が表示されます。

アイテムが消えると、レバーの上下左右（カーソルキーの上下左右）でカーソルが動くようになります。記憶を頼りに、カーソルをアイテムの位置に合わせて、ボタン0（Zキー）を押します。カーソルがアイテムに重なっていると、アイテムを回収することができ、そのアイテムは見えるようになります。

MEMORIZING ITEM POSITION → p. 390

地図を頼りにアイテムを探す

ステージのどこかに隠されているアイテムを、地図を頼りに探し出すアクションです。地図にはアイテムの周囲の地形が表示されます。地図とステージの地形をじっくりと見比べて、アイテムの場所を推理します。

迷路状のステージのどこかに、アイテムが隠されています（Fig. 6-5）。キャラクターはレバー操作で上下左右に動きます。

アイテムは見えませんが、かわりにアイテムの地図が表示されます（Fig. 6-6）。地図にはア

Fig. 6-5 迷路状のステージ

壁
アイテム（見えない）
キャラクター（上下左右に動く）

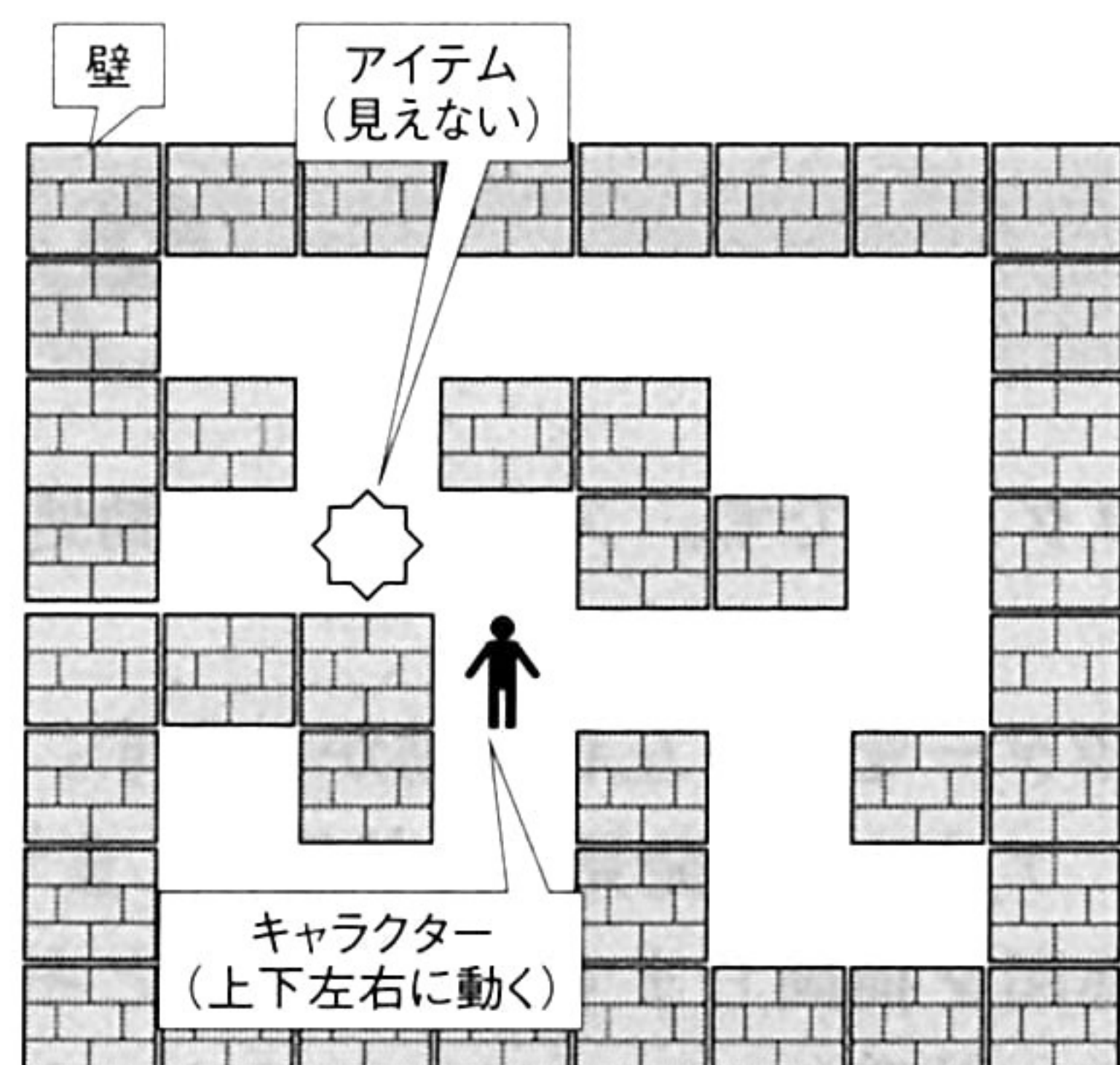
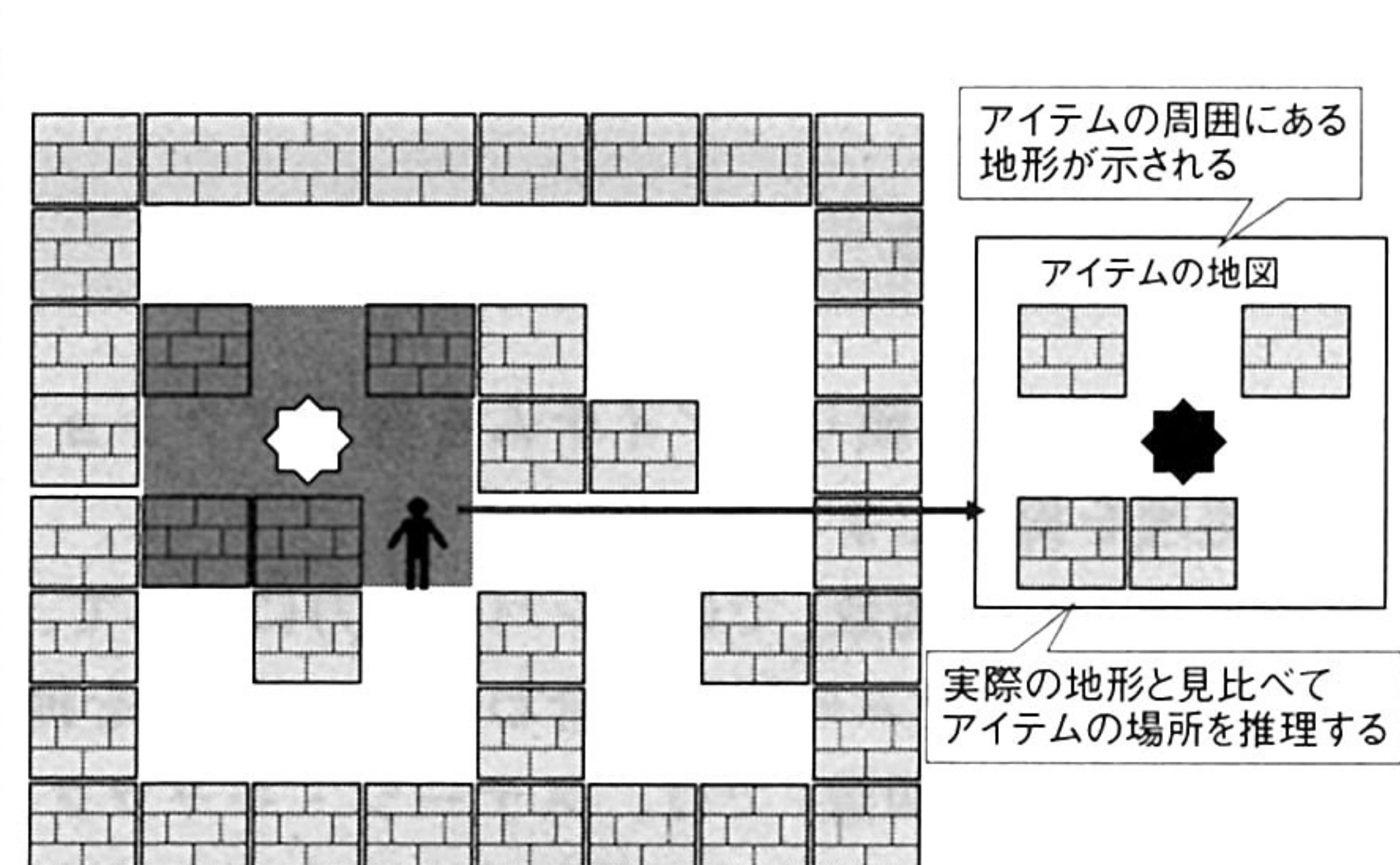



Fig. 6-6 アイテムの地図

アイテムの周囲にある地形が示される
アイテムの地図
実際の地形と見比べてアイテムの場所を推理する

イテムの周囲にある地形が示されているので、実際の地形とよく見比べて、アイテムの場所を推理します。

キャラクターをアイテムの場所に移動し、ボタンを押せば、アイテムを回収することができます。回収したアイテムは見えるようになります。

地図を頼りにアイテムを探すアクションは『シンドバッドミステリー』に採用されています。このゲームでは、まずステージ内に散らばった地図の断片を集める必要があります。断片を集めるごとに、地図が少しずつ見えるようになります。地図と実際の地形をよく見比べて、アイテムの場所を推理し、回収することがゲームの目的です。

アルゴリズム

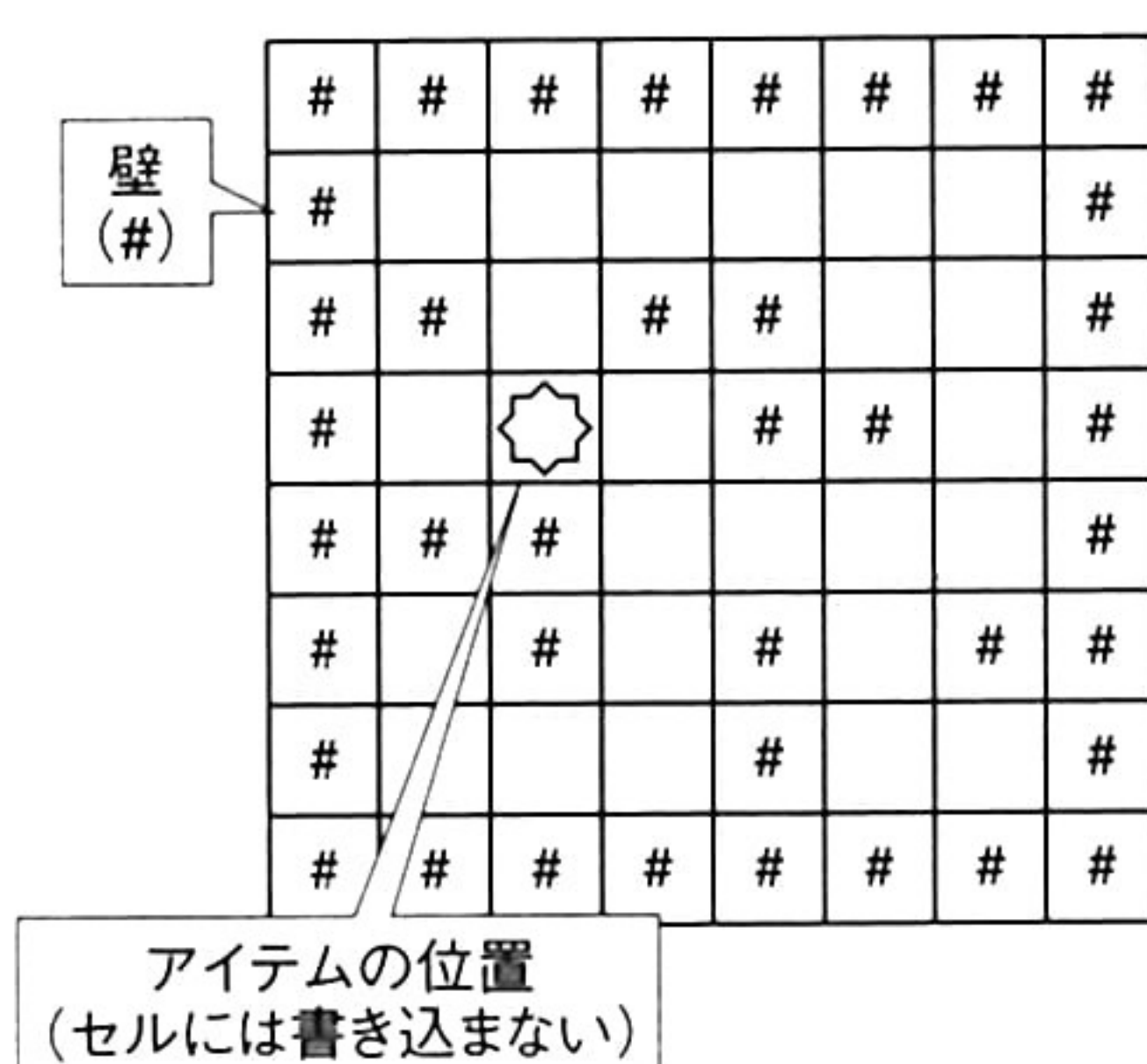
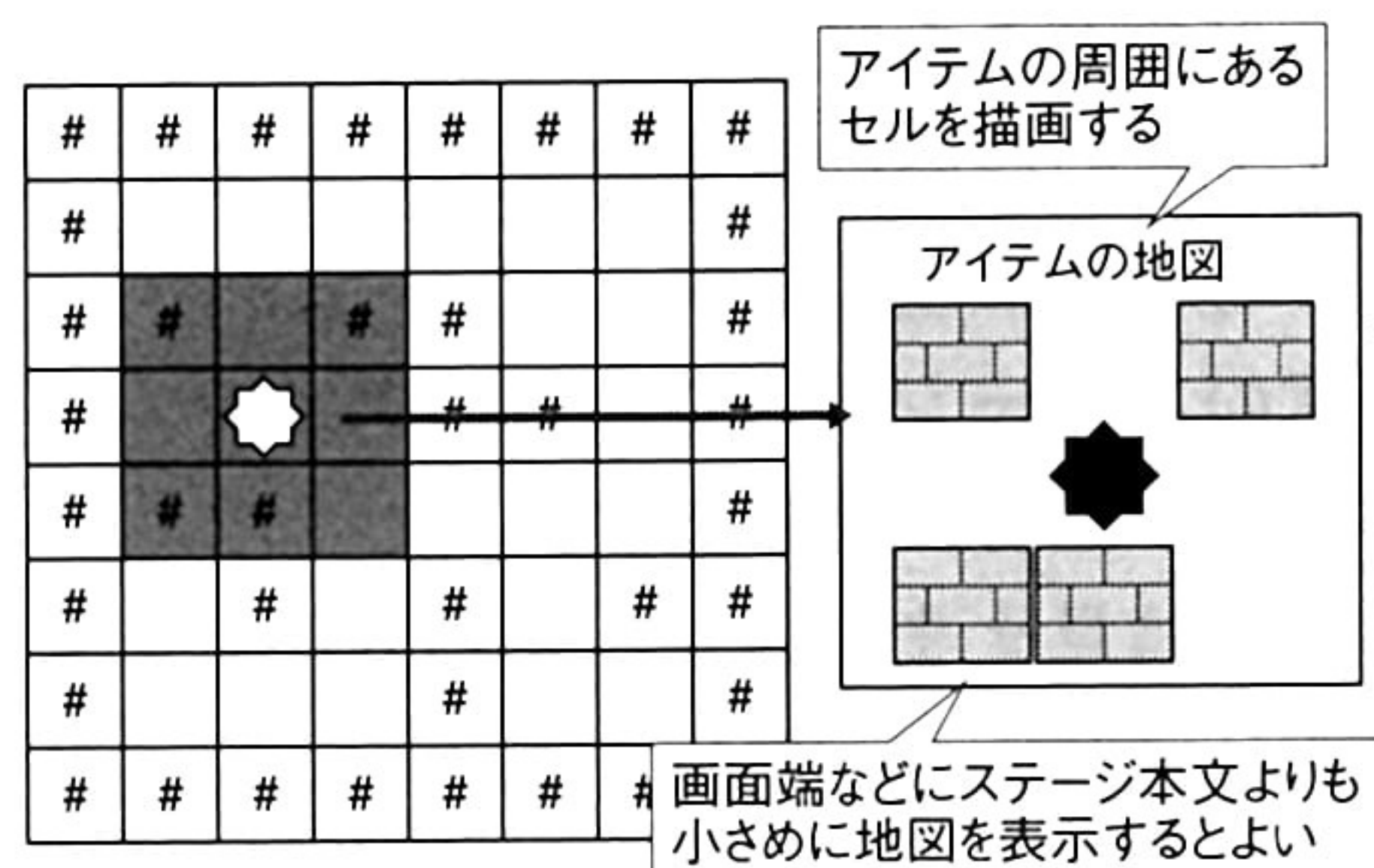
地図を頼りにアイテムを探すアクションを実現するためには、まずステージをセルで表現します（Fig. 6-7）。ステージの壁は「#」で表しました。ステージを描画したり、キャラクターを動かすときには、このセルを調べます。なお、アイテムはセルには書き込みません。

地図を表示するには、アイテムの位置の周囲にあるセルを描画します（Fig. 6-8）。描画方法はステージ本体の描画と同様です。ステージ本体よりも小さめに表示するとよいでしょう。

Fig. 6-7 ステージをセルで表現する

壁（#）
アイテムの位置（セルには書き込まない）

Fig. 6-8 地図の描画

アイテムの周囲にあるセルを描画する
アイテムの地図
画面端などにステージ本文よりも小さめに地図を表示するとよい

プログラム

List 6-2は地図を頼りにアイテムを探すアクションのプログラムです。ステージの移動処理と描画処理を掲載しました。

移動処理（Move関数）では、レバー入力に応じて、キャラクターを上下左右に動かします。キャラクターをアイテムの位置に合わせて、ボタンを押したら、アイテムを発見した状態にします。

描画処理（Draw関数）では、ステージ・キャラクター・地図を描画します。ステージとキャラクターの描画については、特に変わったことはありません。地図については、アイテムを中

心として、アイテムの周囲にある地形を描画します。ステージ本体よりも小さめに表示すると、地図らしい雰囲気が出るでしょう。

本書のサンプルでは最初から地図が表示されていますが、地図の断片を拾うごとに、地図が少しずつ見えていくというのも面白い仕掛けです。この場合は、ステージに地図の断片を配置しておき、キャラクターが断片を拾うたびに、地図を表示する範囲を広げます。

List 6-2 地図を頼りにアイテムを探す(CTreasureMapStageクラス)

```
// 移動処理
bool CTreasureMapStage::Move(const CInputState* is) {

    // レバー入力に応じて、キャラクターを上下左右に動かす
    if (!PrevLever) {
        int cx=CX, cy=CY;
        if (is->Left) cx--;
        if (is->Right) cx++;
        if (is->Up) cy--;
        if (is->Down) cy++;

        // 移動先が壁でなければ、キャラクターを動かす
        if (Cell->Get(cx, cy)!='#') {
            CX=cx;
            CY=cy;
        }
    }
    PrevLever=is->Left||is->Right||is->Up||is->Down;

    // キャラクターをアイテムの位置に合わせてボタンを押したら、
    // アイテムを発見した状態にする
    if (
        !PrevButton && is->Button[0] &&
        CX==IX && CY==IY
    ) {
        State=1;
    }
    PrevButton=is->Button[0]||is->Button[1];

    return true;
}

// 描画処理
void CTreasureMapStage::Draw() {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // 解像度に応じて描画サイズを変える
    float
```

```
    sw=Game->GetGraphics()->GetWidth()/xs,
    sh=Game->GetGraphics()->GetHeight()/ys;
// ステージ内のすべてのセルを描画する
for (int y=0; y<ys; y++) {
    for (int x=0; x<xs; x++) {

        // 壁を描画する
        if (Cell->Get(x, y)=='#') {
            Game->Texture[TEX_FLOOR]->Draw(
                x*sw, y*sh, sw, sh,
                0, 0, 1, 1, COL_BLACK);
        }
    }
}

// キャラクターを描画する
Game->Texture[TEX_MAN]->Draw(
    CX*sw, CY*sh, sw, sh,
    0, 0, 1, 1, COL_BLACK);

// アイテムが発見ずみならば、アイテムを描画する
if (State==1) {
    Game->Texture[TEX_ITEM]->Draw(
        IX*sw, IY*sh, sw, sh,
        0, 0, 1, 1, COL_BLACK);
}

// 地図を表示する
for (int y=-2; y<3; y++) {
    for (int x=-4; x<4; x++) {

        // グラフィックの描画座標
        // 地図はステージよりも小さく表示する
        float dx=(x+16)*sw/2, dy=(y+3)*sh/2;

        // 壁を描画する
        if (Cell->Get(IX+x, IY+y)=='#') {
            Game->Texture[TEX_FLOOR]->Draw(
                dx, dy, sw/2, sh/2,
                0, 0, 1, 1, COL_BLACK);
        }

        // アイテムを描画する
        if (x==0 && y==0) {
            Game->Texture[TEX_ITEM]->Draw(
                dx, dy, sw/2, sh/2,
                0, 0, 1, 1, COL_BLACK);
        }
```

```
            }
        }
    }
```

SAMPLE

「TREASURE MAP」は「地図を頼りにアイテムを探す」のサンプルです。

レバーの上下左右（カーソルキーの上下左右）でキャラクターが動きます。画面上方に表示されている地図をヒントに、アイテムの場所を推理します。アイテムのある場所に移動して、ボタン0（Zキー）を押すと、アイテムを回収することができます。回収したアイテムは、見えるようになります。

TREASURE MAP → p. 390

荷物を指定の場所に運ぶ

迷路のなかに配置された荷物を、指定の場所まで運ぶアクションです。荷物にはいくつかの種類があり、種類ごとに目的地が決まっています。

迷路のなかに数種類の荷物が配置されています（Fig. 6-9）。荷物の種類に対応して、荷物の運び先も用意されています。プレイヤーはレバー操作でカーソルを動かして、荷物を目的地まで運びます。

カーソルを荷物に重ねて、ボタンを押すと、荷物を拾うことができます（Fig. 6-10）。荷物を持った状態でボタンを押すと、荷物を置きます（Fig. 6-11）。

荷物を持った状態で、他の荷物の上でボタンを押すと、持っている荷物と置いてある荷物を

Fig. 6-9 ステージの構成

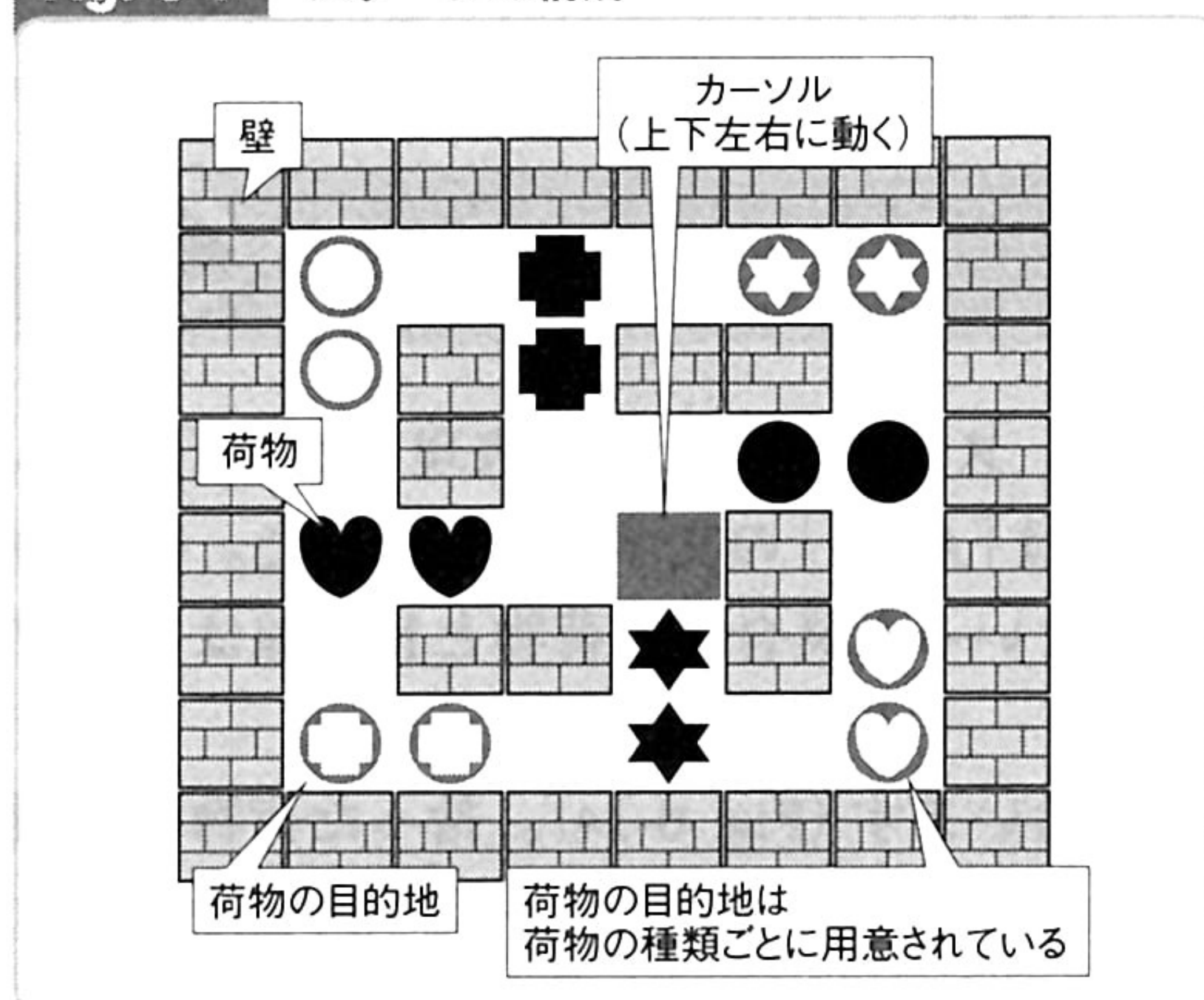



Fig. 6-10 荷物を拾う

Fig. 6-11 荷物を置く

荷物を持った状態で
ボタンを押すと荷物を置く

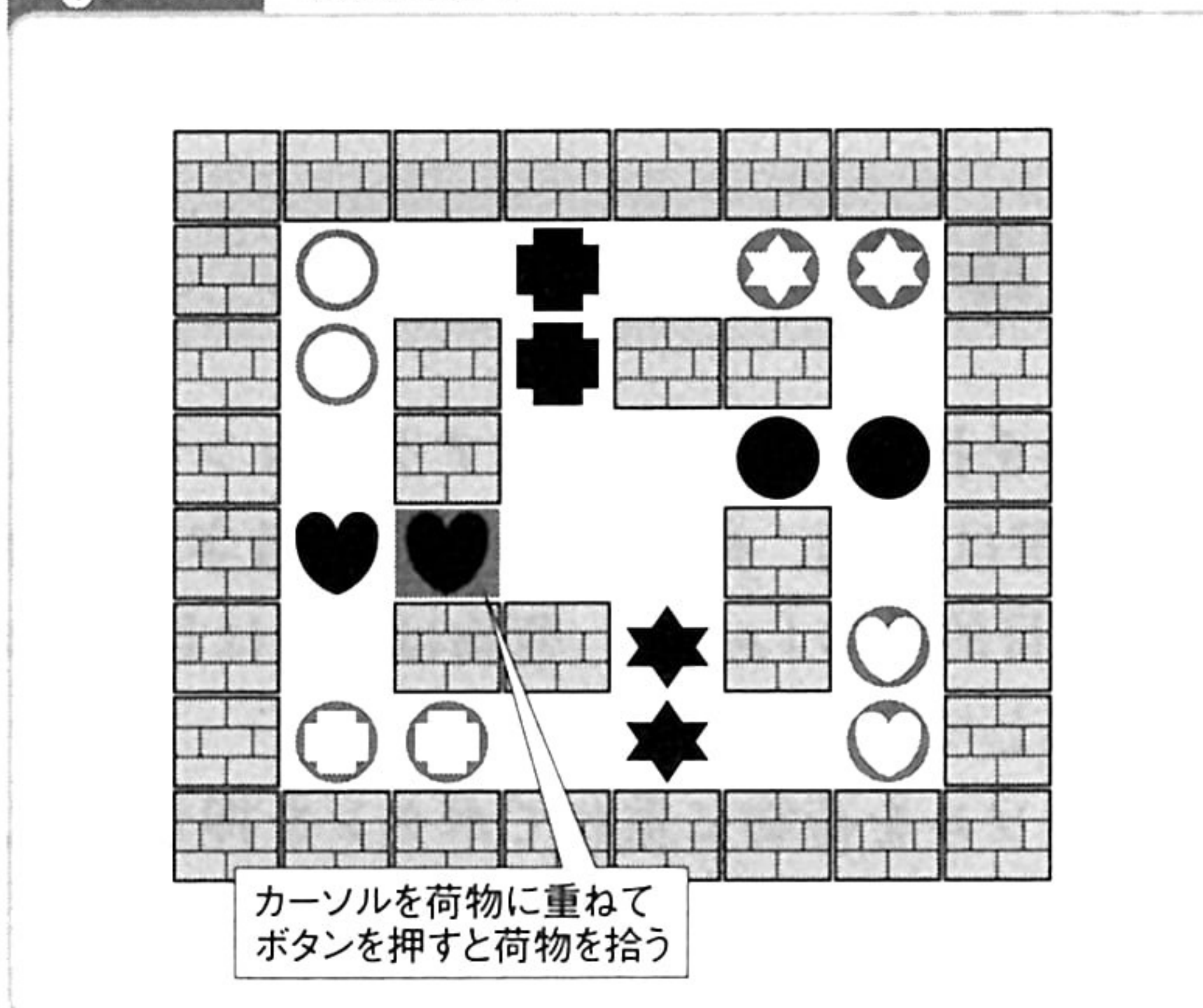
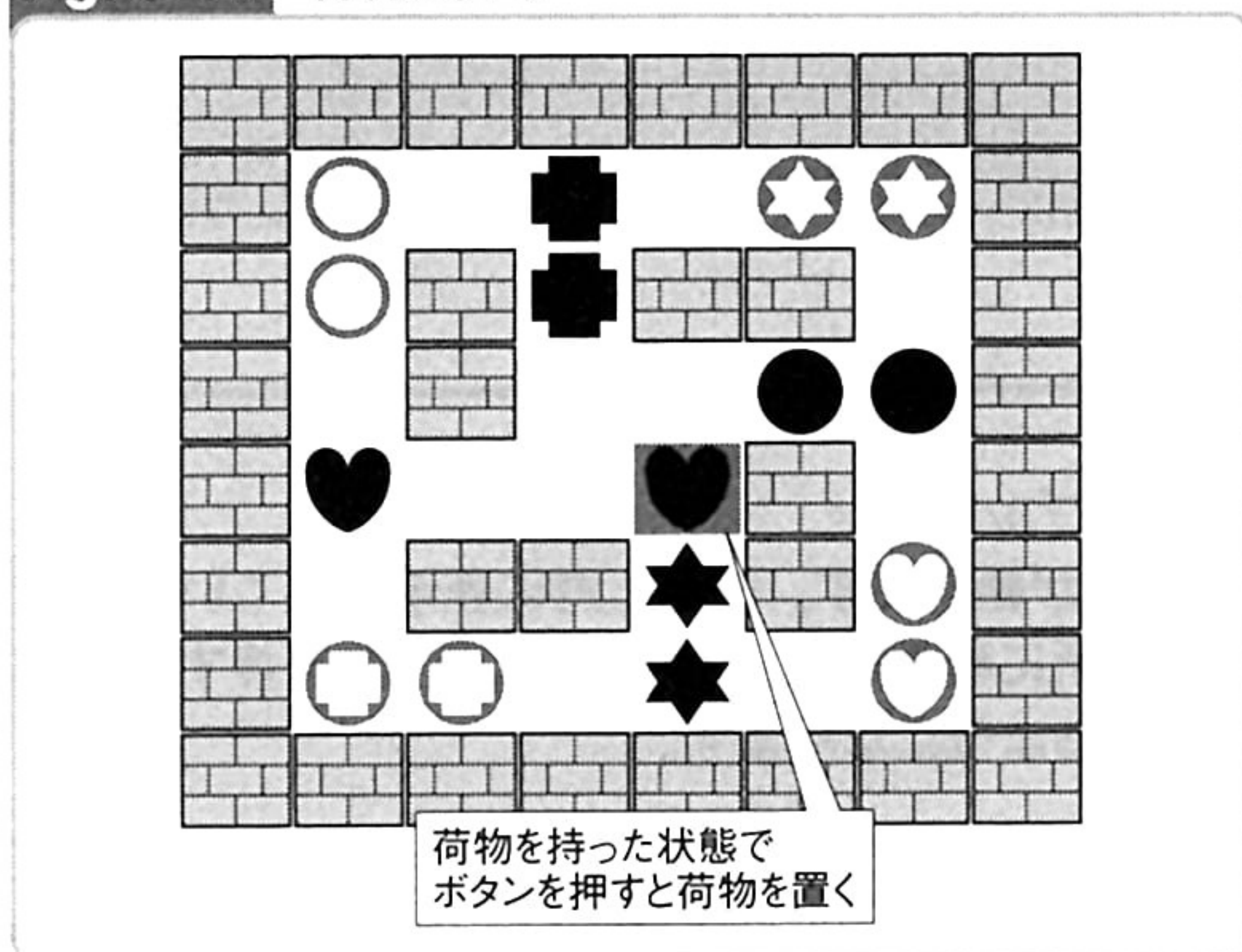
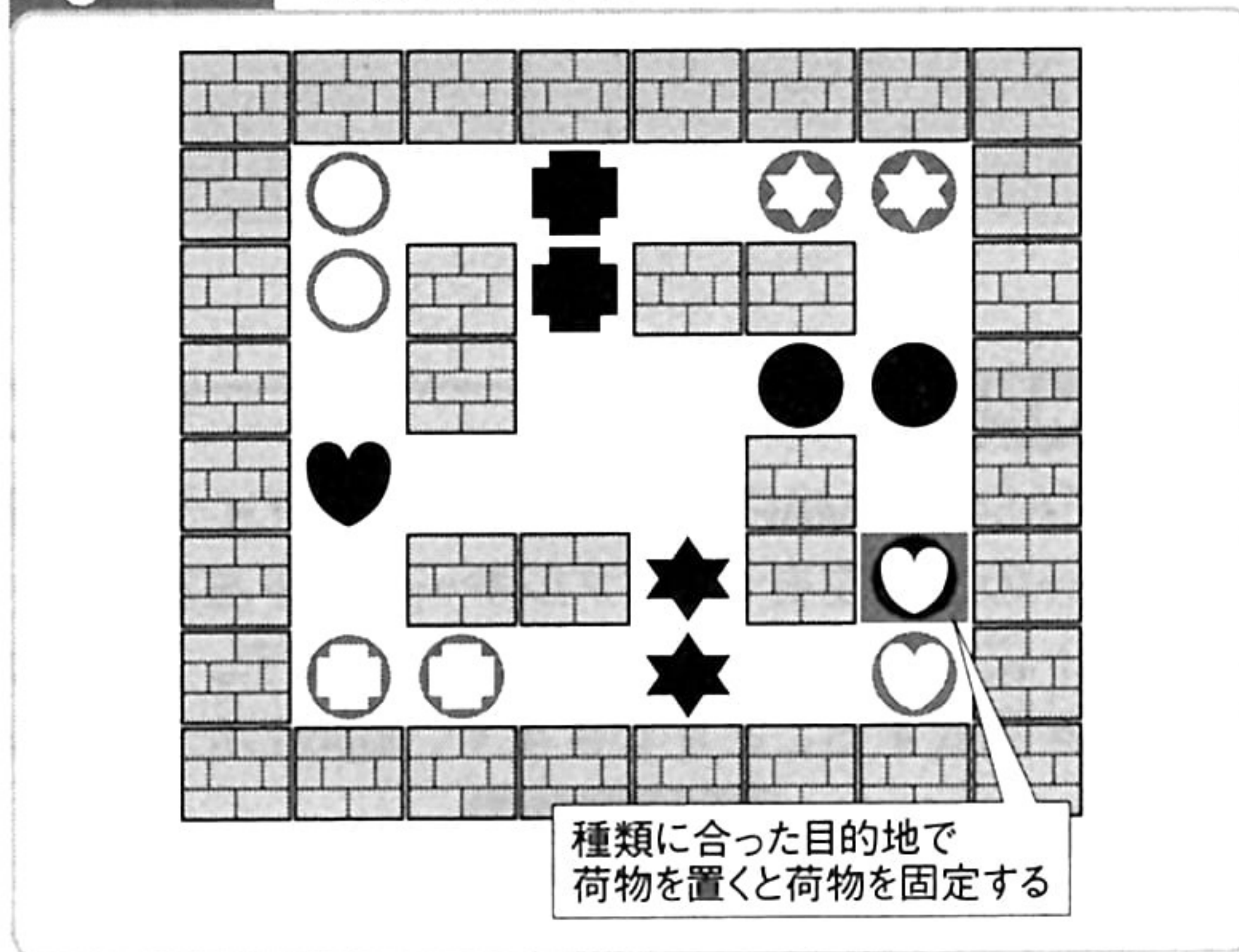



Fig. 6-12 荷物を固定する

種類に合った目的地で
荷物を置くと荷物を固定する

入れ替えます。これは「ボールを入れ替える」(→p. 331) に似ています。

荷物の種類に合った目的地で荷物を置くと、荷物を固定することができます (Fig. 6-12)。すべての荷物を正しい目的地に運ぶことが、ゲームの目的です。

荷物を指定の場所に運ぶゲームには『ピッキン』があります。このゲームでは、荷物に何種類かの色があり、目的地も色別に用意されています。キャラクターを動かし、荷物を拾って、正しい目的地に集めることがゲームの目的です。

他の荷物のある場所は、荷物を通すことができないので、荷物を運ぶ順序をよく考える必要があります。また、敵が迷路内を動き回っていて、ときどき勝手に荷物を運んでしまいます。

「荷物を押す」(→p. 19) で紹介した『倉庫番』も、荷物を指定の場所に運ぶゲームです。このゲームの場合には、キャラクターを動かし、荷物を押して運びます。荷物を引くことはできず、押すだけなので、荷物を運ぶ順序や経路をよく考える必要があります。

「重力で落ちる荷物を押す」(→p. 30) で紹介した『フラッピー』も、荷物を指定の場所に運ぶゲームです。このゲームでは、『倉庫番』と同じく荷物を押して運びますが、荷物が重力で落下する点が異なります。荷物を谷間に積んで橋を作り、別の荷物を渡すといったアクションも楽しめます。

アルゴリズム

荷物を指定の場所に運ぶアクションを実現するには、ステージをセルで表現します (Fig. 6-13)。壁は「#」、荷物は「0〜3」の数字、荷物の目的地は「A〜D」の文字で表しました。荷物の「0」は目的地の「A」に、荷物は「1」は目的地の「B」にといった具合に、荷物と目的地はそれぞれ対応しています。

カーソルを荷物に重ねてボタンを押したら、荷物を拾います (Fig. 6-14)。拾った荷物の種類を記録して、セルを空にします。

Fig. 6-13 ステージをセルで表現する

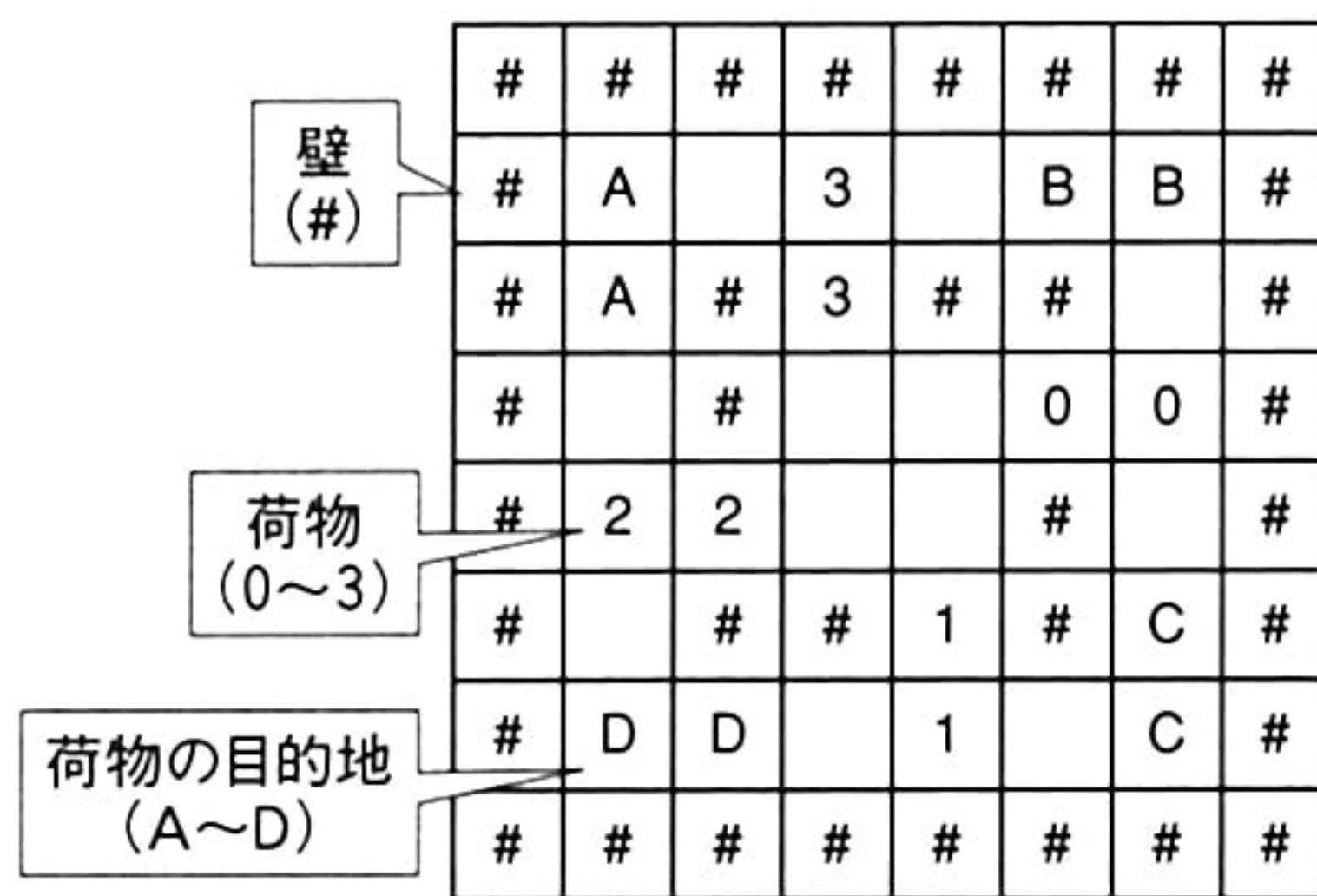



Fig. 6-14 荷物をセルから拾う

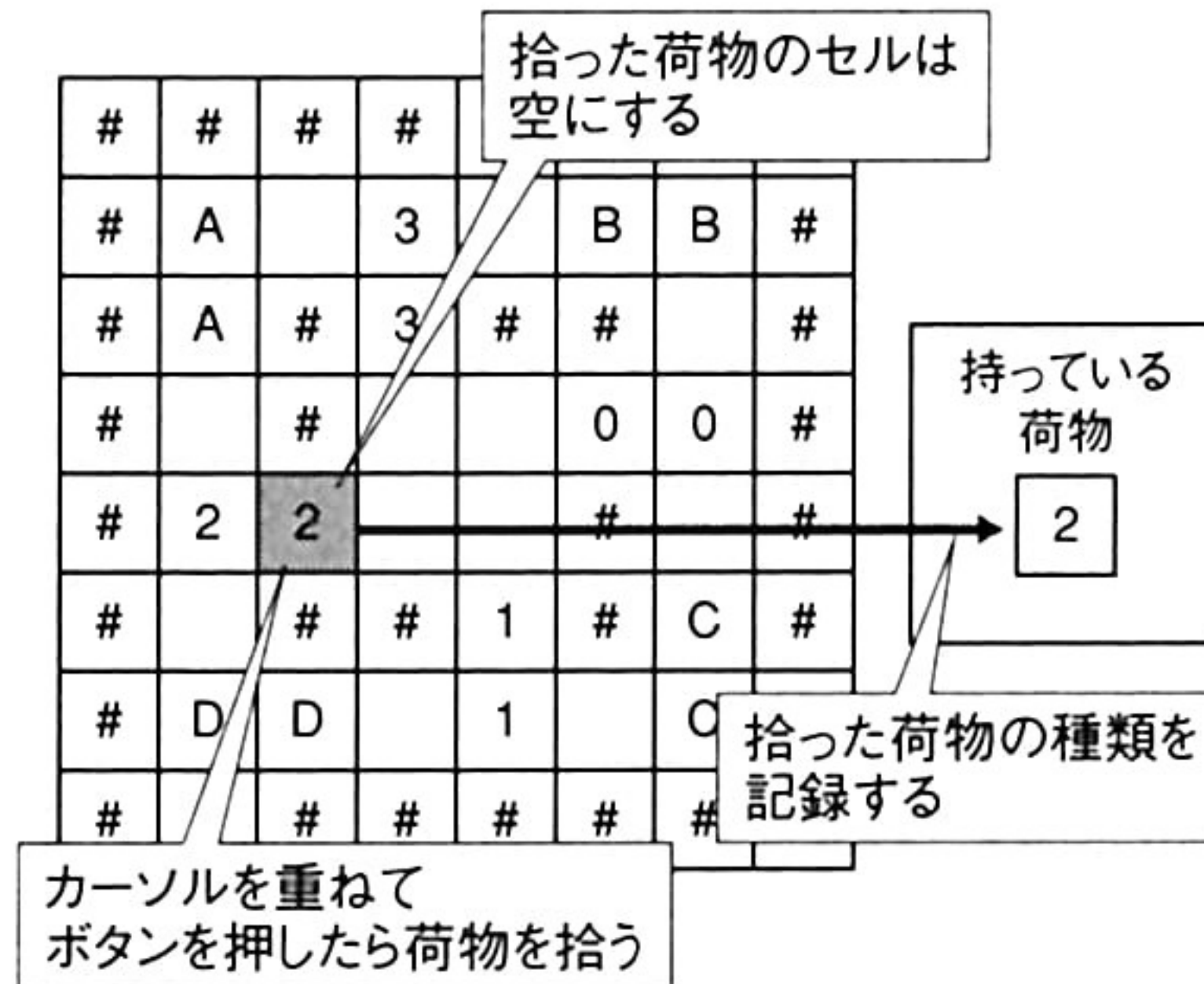



Fig. 6-15 荷物をセルに置く

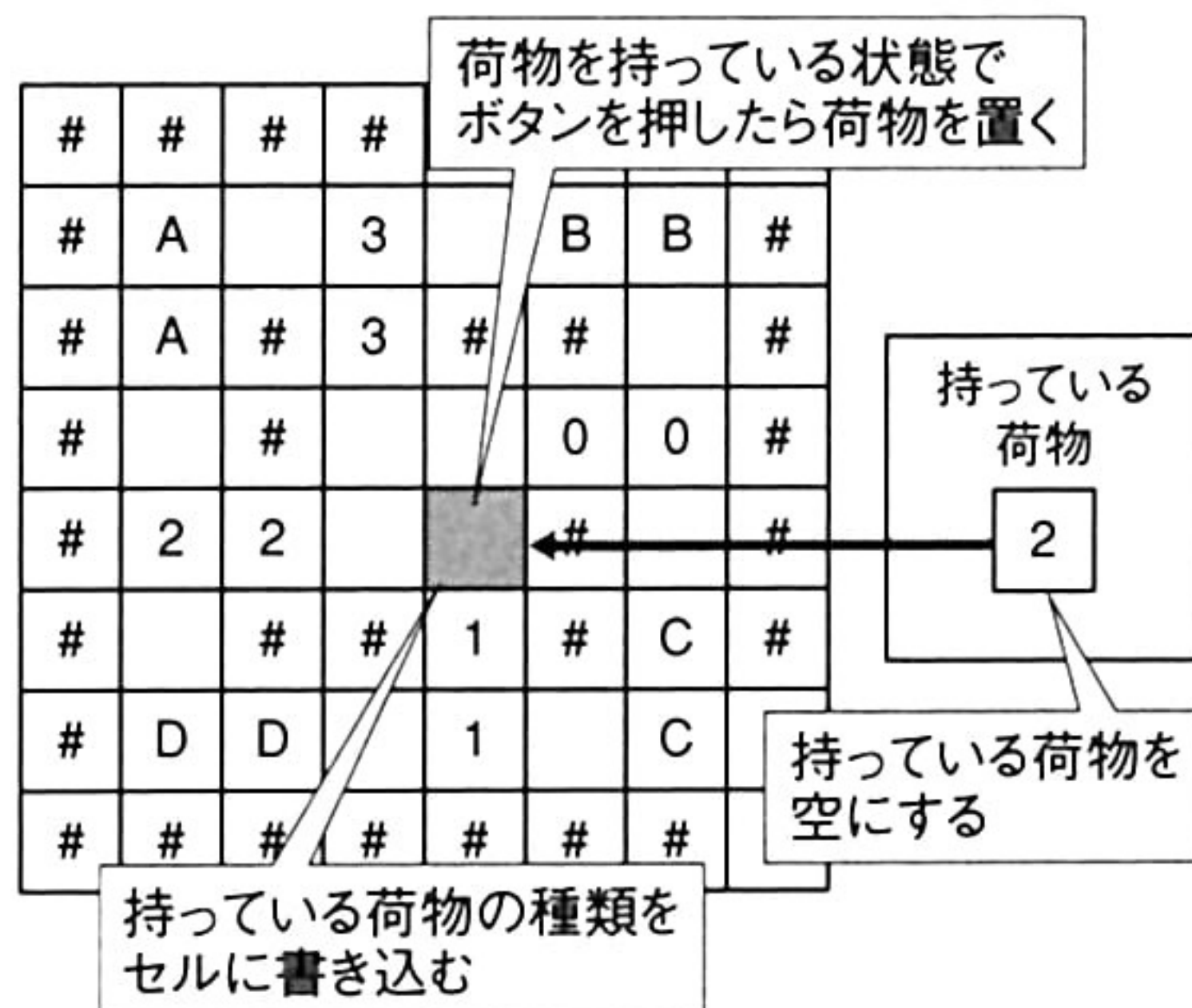
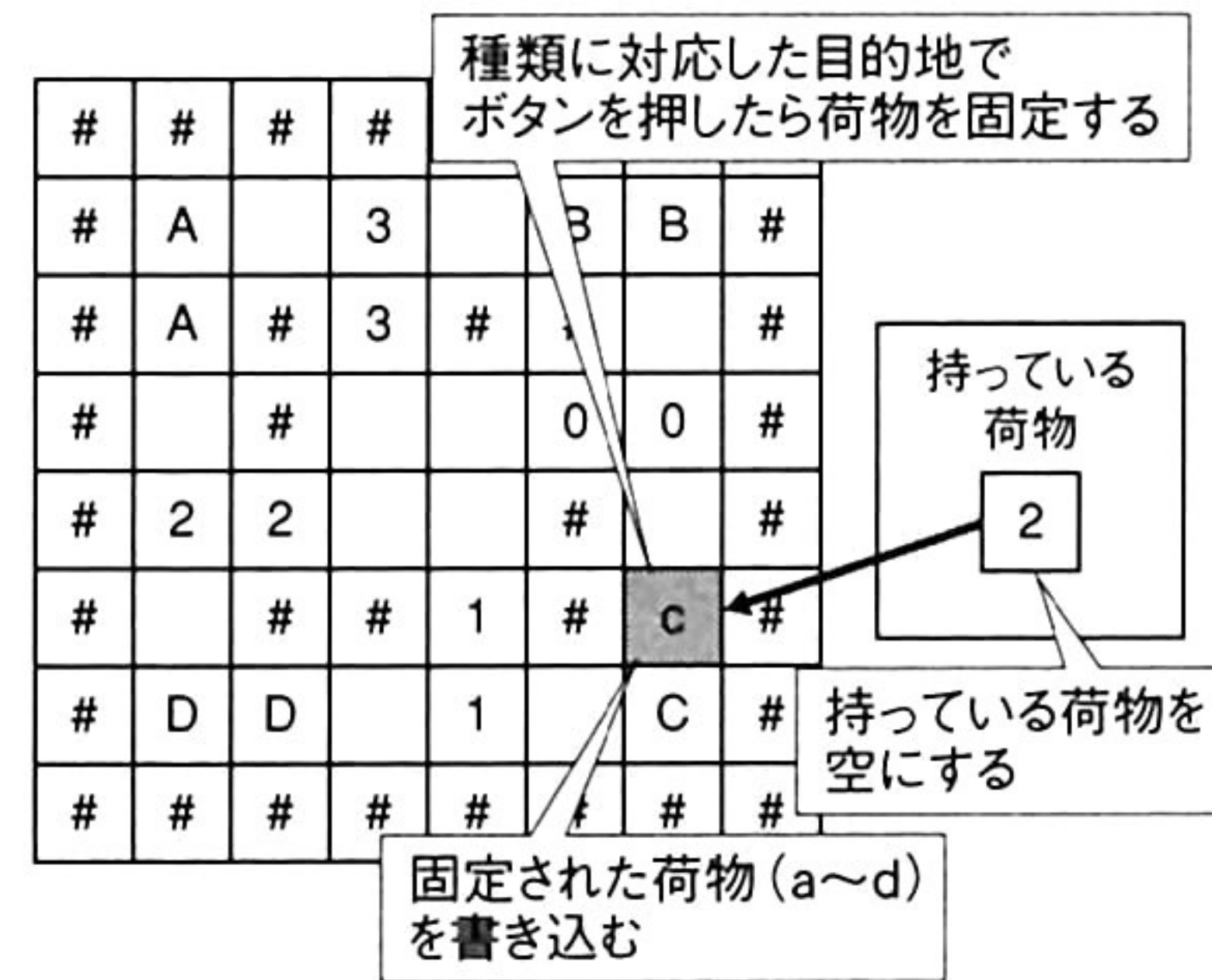



Fig. 6-16 荷物をセルに固定する

荷物を持っている状態でボタンを押したら、荷物を置きます(Fig. 6-15)。セルに荷物の種類を書き込み、持っている荷物を空にします。もしもその位置に荷物が置いてあったら、持っている荷物と、置いてある荷物とを入れ替えます。

荷物の種類に対応した目的地の上でボタンを押したら、荷物を固定します(Fig. 6-16)。ここでは固定された荷物を「a〜d」の文字で表します。セルに荷物の種類に応じてa〜dの文字を書き込み、持っている荷物を空にします。

プログラム

List 6-3は荷物を指定の場所に運ぶプログラムです。ステージの移動処理を掲載しました。

移動処理では、レバー入力に応じて、カーソルを上下左右に動かします。ボタンを押したら、カーソルの位置にあるセルの種類を調べます。

セルが空または荷物ならば、持っている荷物と置いてある荷物を入れ替えます。セルが持っている荷物の目的地ならば、そこに荷物を固定します。

List 6-3 荷物を指定の場所に運ぶ(CDeliveringLoadStageクラス)

```
// 移動処理
bool CDeliveringLoadStage::Move(const CInputState* is) {

    // レバー入力に応じて、カーソルを上下左右に動かす
    if (!PrevLever) {
        int cx=CX, cy=CY;
        if (is->Left) cx--;
        if (is->Right) cx++;
        if (is->Up) cy--;
        if (is->Down) cy++;

        // 移動先が壁でなければ、カーソルを移動する
        if (Cell->Get(cx, cy)!='#') {
            CX=cx;
            CY=cy;
        }
    }
    PrevLever=is->Left||is->Right||is->Up||is->Down;

    // ボタンを押したときの処理
    if (!PrevButton && is->Button[0]) {

        // カーソルの位置にあるセルの種類を取得する
        char c=Cell->Get(CX, CY);

        // セルが空または荷物ならば、
        // 持っている荷物と置いてある荷物を入れ替える
        if (
            c==' ' ||
            ('0'<=c && c<'0'+DELIVERING_LOAD_TYPE)
        ) {
            // 持っている荷物を置く
            Cell->Set(CX, CY, Type);

            // 置いてある荷物を持つ
            Type=c;
        } else

        // セルが持っている荷物の目的地ならば、
        // 荷物を固定する
        if (
            'A'<=c && c<'A'+DELIVERING_LOAD_TYPE &&
            Type-'0'==c-'A'
        ) {
            // 荷物を固定する
            Cell->Set(CX, CY, 'a'+Type-'0');

            // 持っている荷物を空にする
```

```
            Type=' ';
        }
    }
    PrevButton=is->Button[0]||is->Button[1];

    return true;
}
```

SAMPLE

「DELIVERING LOAD」は「荷物を指定の場所に運ぶ」のサンプルです。

レバーの上下左右（カーソルキーの上下左右）でカーソルが動きます。ボタン0（Zキー）を押すと、カーソルの位置にある荷物を拾います。荷物を持っているときにボタン0を押すと、持っている荷物と、カーソルの位置にある荷物とを入れ替えます。

荷物には種類ごとに目的地があります。対応する目的地まで荷物を運んで、ボタン0を押すと、荷物を固定することができます。

DELIVERING LOAD → p. 390

床を作って進む

床を作り、その上に飛び乗ることによって、ステージを登っていくアクションです。床を上手に作りながら、ステージ内のいろいろな場所に配置されたアイテムを集め、ステージから脱出することがゲームの目的です。

キャラクターはレバー操作で左右に動きます（Fig. 6-17）。ボタンを押すと、キャラクターの進行方向に床を作ることができます。

床がある方向に移動すると、キャラクターは床の上に飛び乗ります（Fig. 6-18）。ただし、キャラクターの上や床の上に障害物があるときには、飛び乗ることができません。

Fig. 6-17 床を作る

壁
キャラクター（左右に動く）
ボタンを押すと進行方向に床を作ることができる

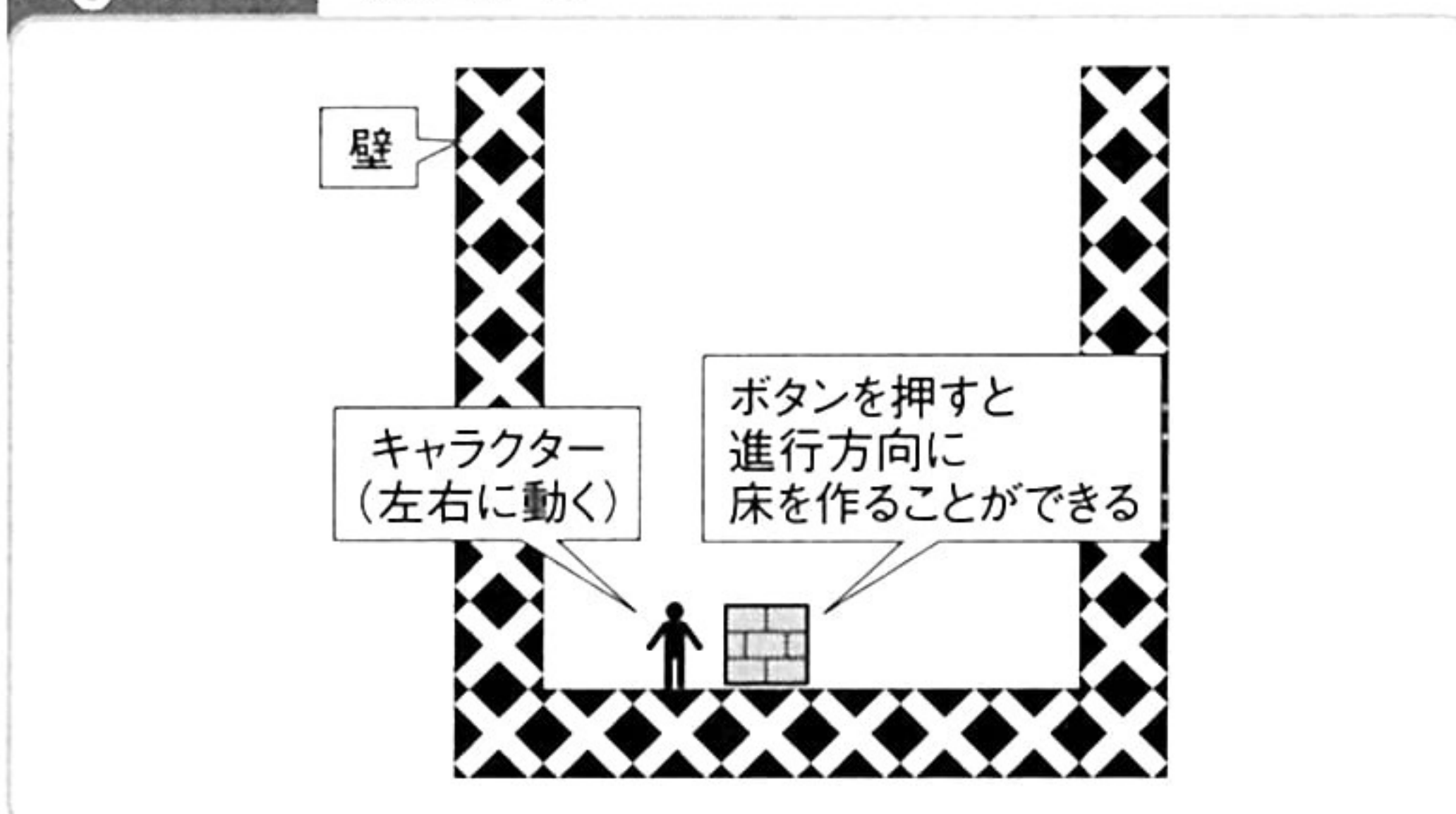



Fig. 6-18 床に乗る

床に向かって移動する
床に飛び乗る

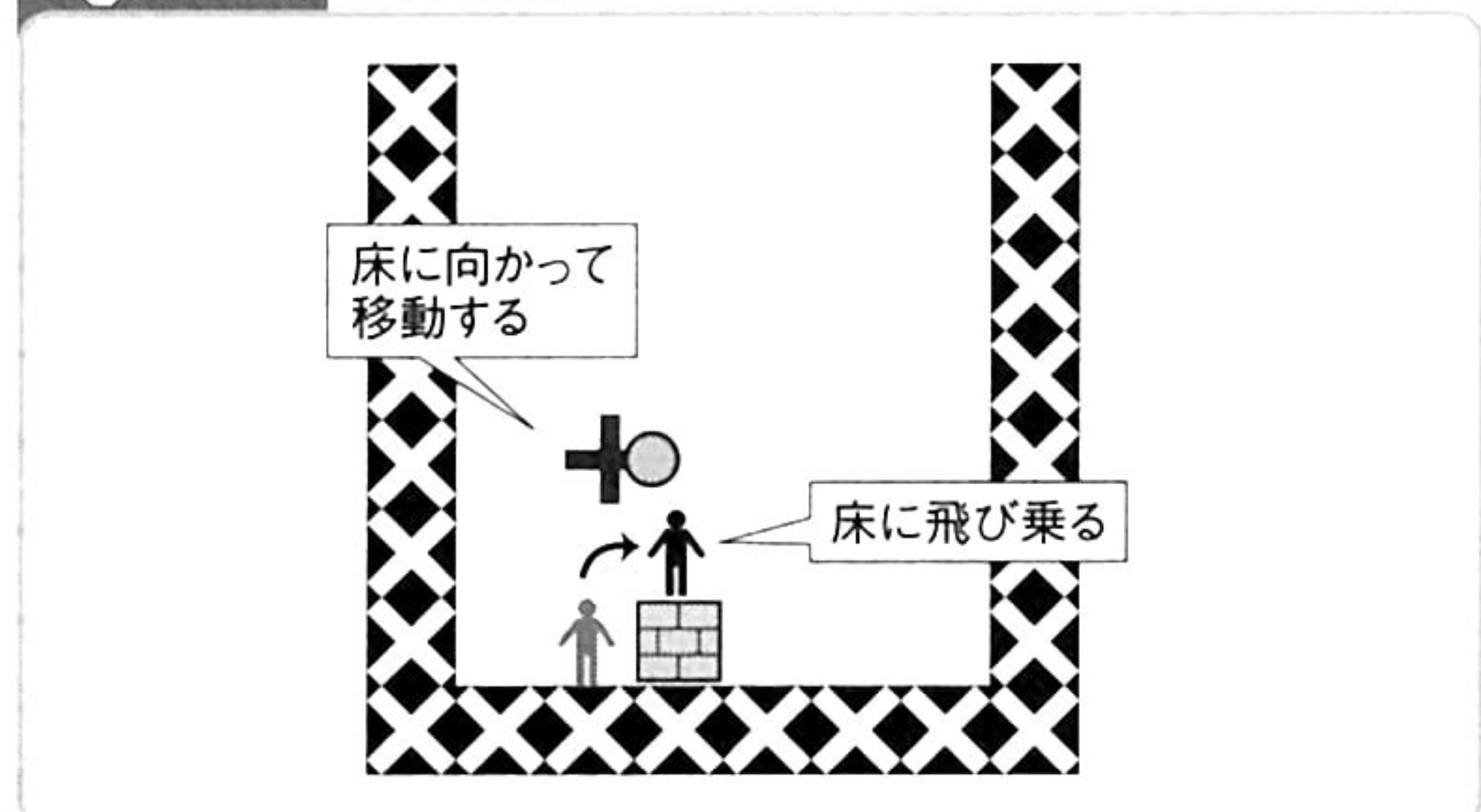



Fig. 6-19 床を消す

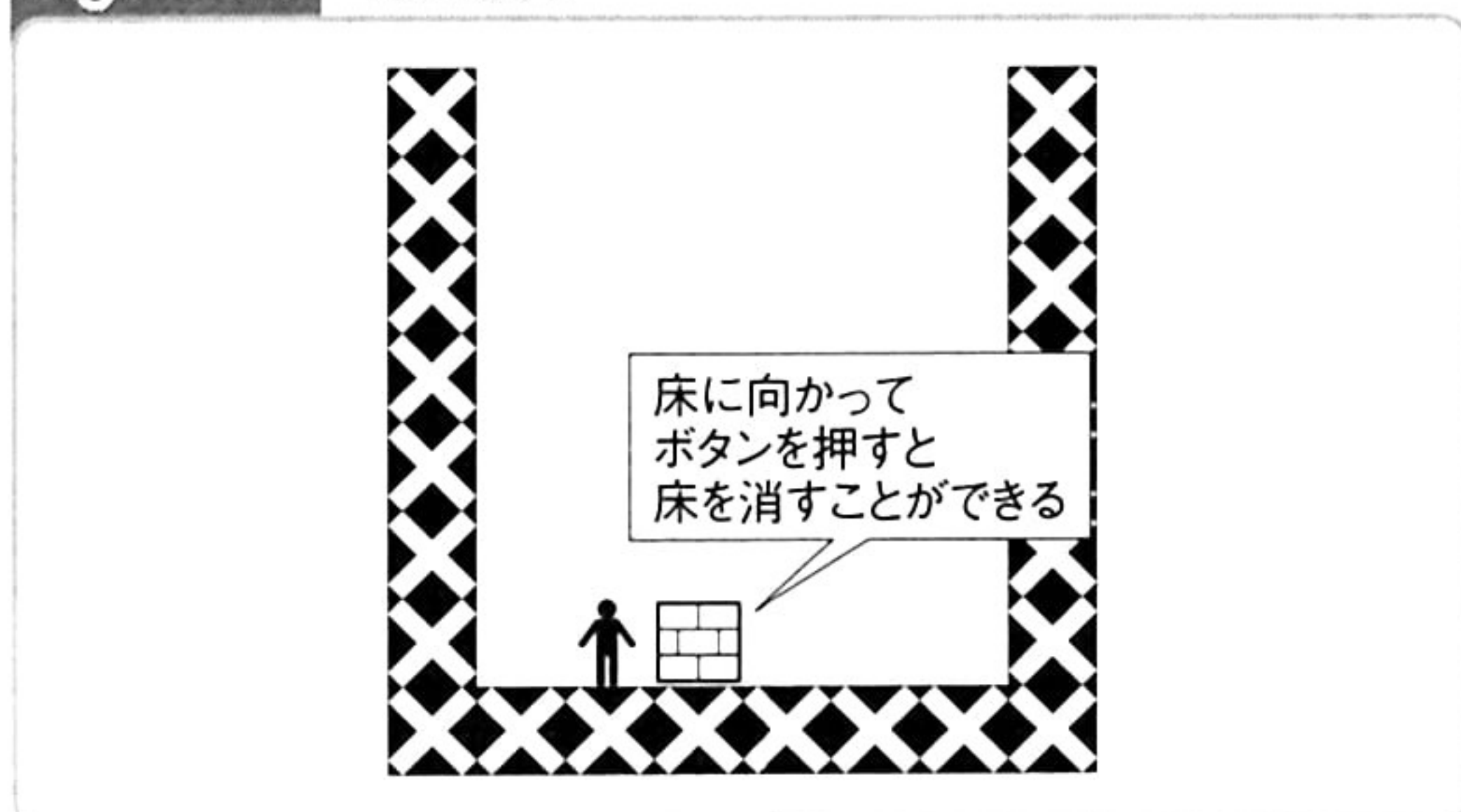
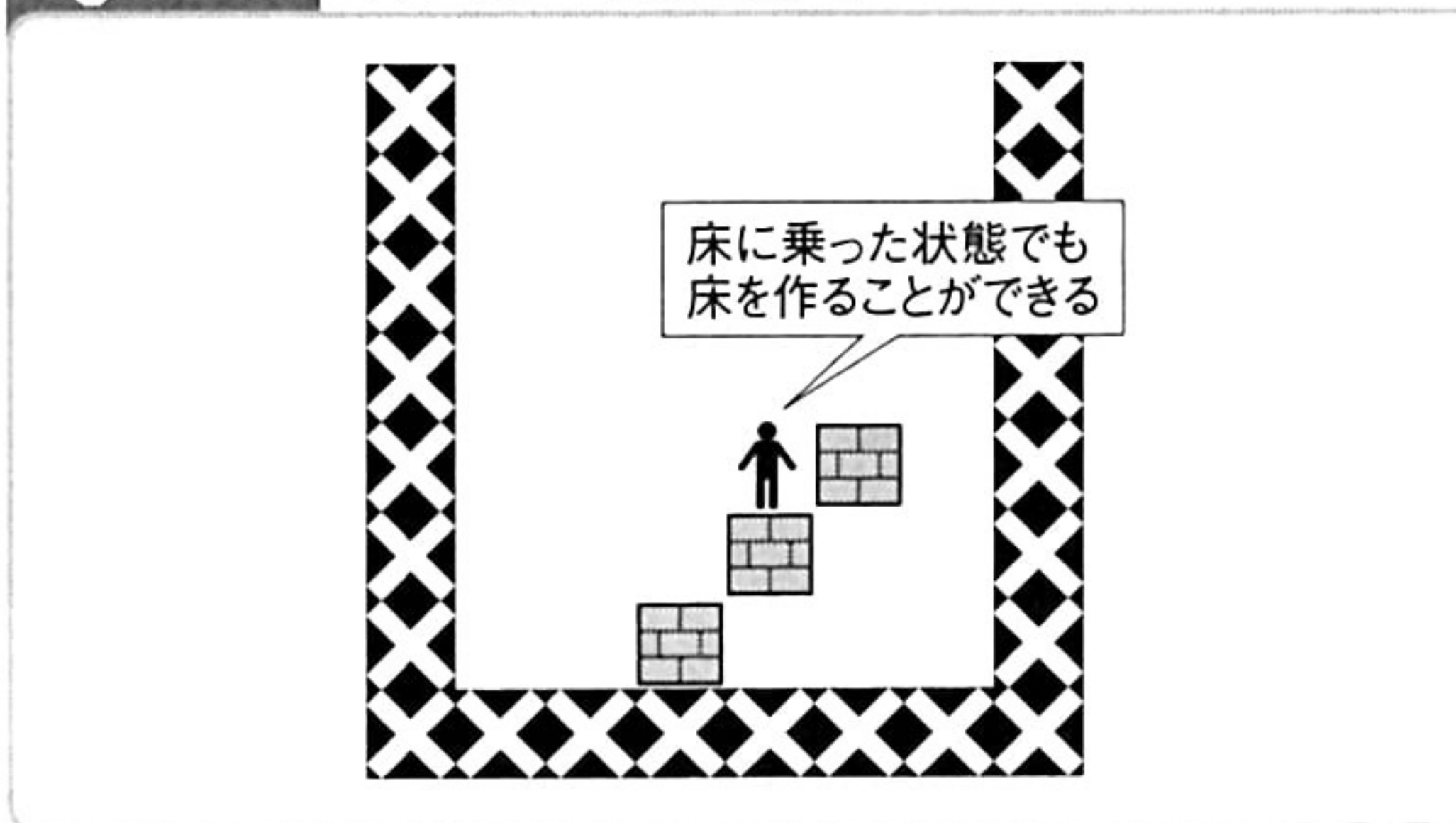



Fig. 6-20 床に乗って床を作る

不要な床は消すこともできます（Fig. 6-19）。床に向かってボタンを押すと、床を消します。

床に乗った状態でも、床を作ることができます（Fig. 6-20）。床に乗ったり、床を作ったりを繰り返すと、ステージの高いところまで登ることができます。

床を作って進むアクションは『ソロモンの鍵』に採用されています。このゲームでは、敵を避けながら、アイテムを回収し、ステージから脱出することが目的です。床を作ってステージを登ったり、床で囲んで敵を閉じ込めたり、敵の下にある床を壊して落としたりと、いろいろなアクションが楽しめます。

アルゴリズム

床を作って進むアクションを実現するには、まずステージをセルで表現します（Fig. 6-21）。ステージの壁は「＝」、キャラクターが作る床は「＃」で表しました。

レバー入力に応じて、キャラクターを左右に動かします。キャラクターが最後に進んだ方向を、進行方向として記録しておきます。

ボタンを押したら、進行方向のセルを調べます（Fig. 6-22）。セルが空ならば、床を作ります。セルが床ならば、セルを空にして、床を消します。

レバーを入力したとき、進行方向に床があるときには、床に飛び乗れるかどうかを調べます（Fig. 6-23）。床の上のセルと、キャラクターの上のセルが、いずれも空の場合には、床に飛び乗ることができます。

Fig. 6-21 ステージをセルで表現する

＝	＝	＝	＝	＝	＝	＝	＝
＝							＝
＝							＝
＝							＝
＝					#		＝
＝				#			＝
＝			#				＝
＝	＝	＝	＝	＝	＝	＝	＝

Fig. 6-22 ボタンを押したときの処理

進行方向のセルが
空のときにボタンを押す

床を作る

進行方向のセルが
床のときにボタンを押す

床を消す

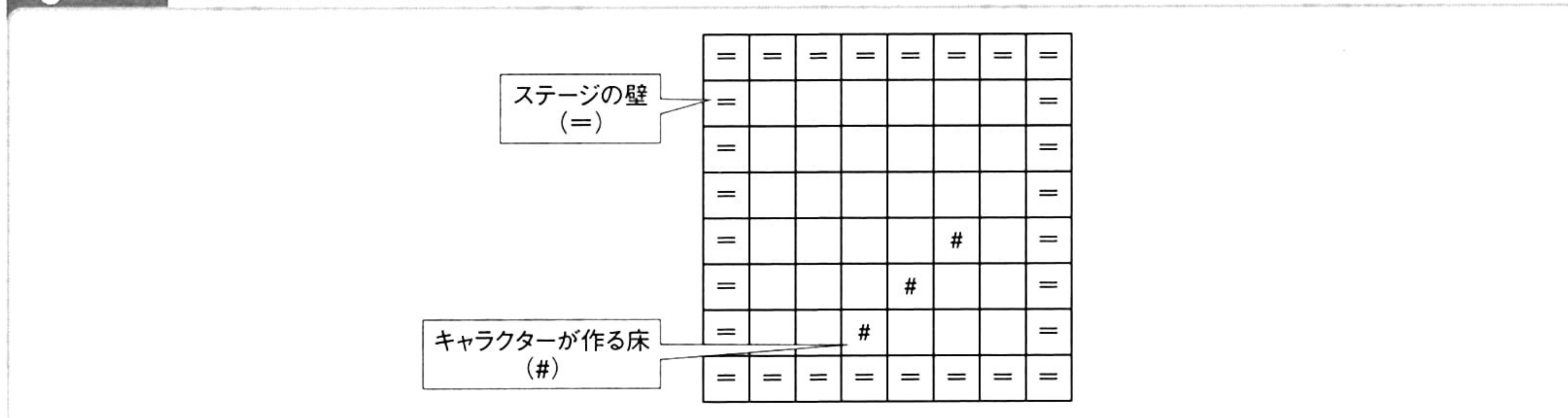
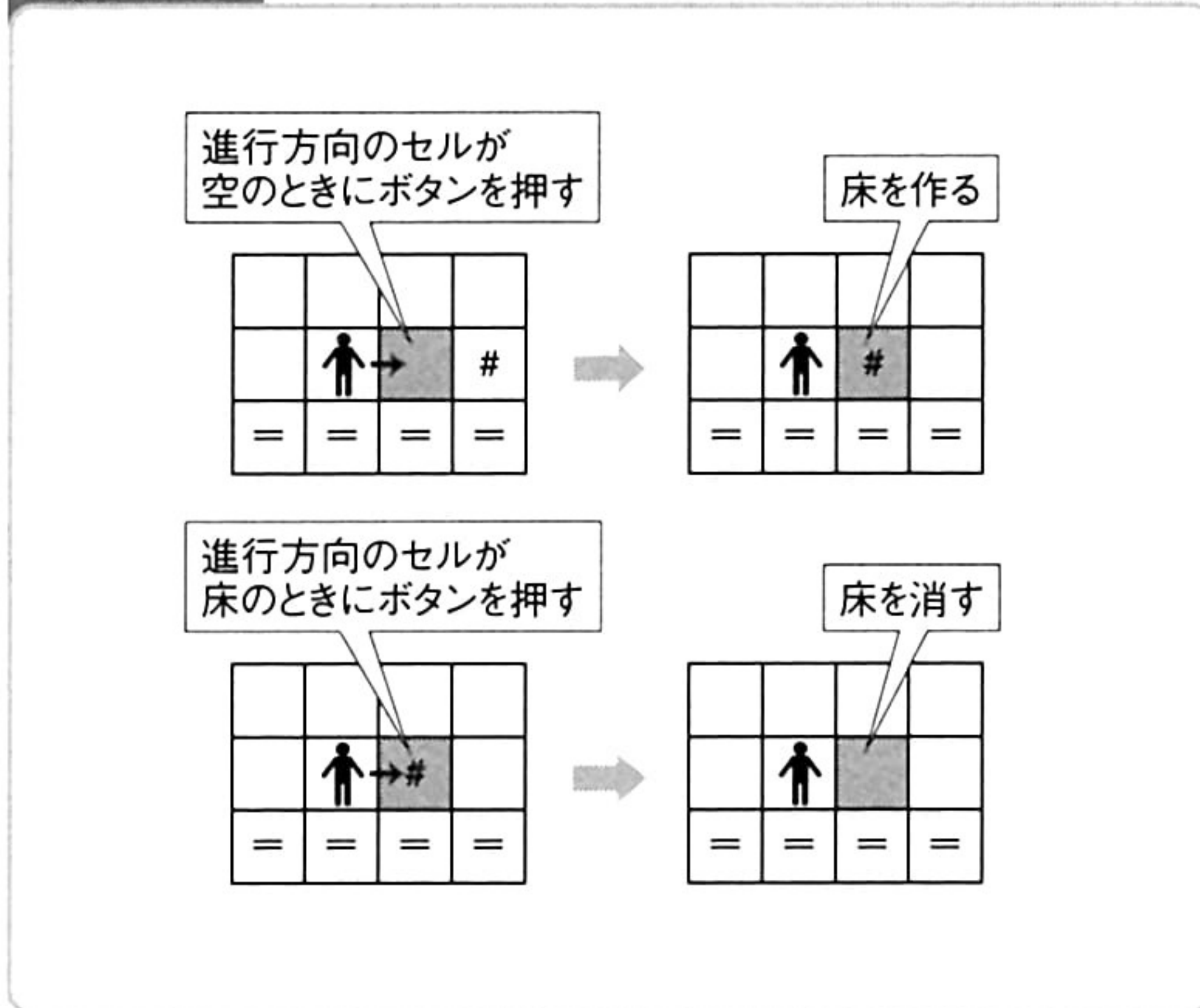
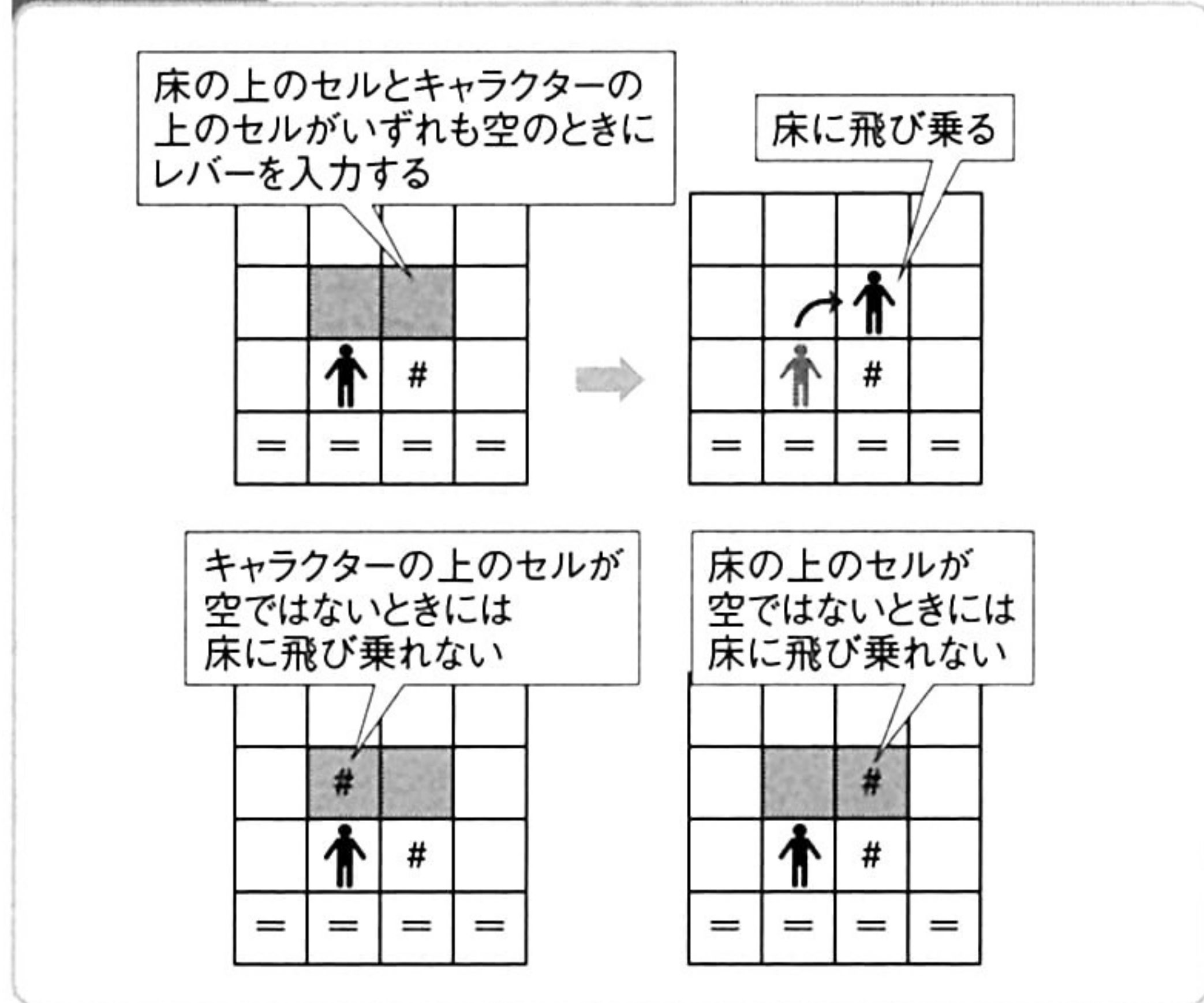



Fig. 6-23 床に飛び乗れるかどうかを調べる

床の上のセルとキャラクターの
上のセルがいずれも空のときに
レバーを入力する

床に飛び乗る

キャラクターの上のセルが
空ではないときには
床に飛び乗れない

床の上のセルが
空ではないときには
床に飛び乗れない

プログラム

List 6-4は床を作って進むプログラムです。ステージの移動処理を掲載しました。

移動処理では、まずキャラクターの下のセルを調べます。下のセルが空のときには、キャラクターを落下させます。

次に、レバー入力に応じて、キャラクターを左右に動かします。進行方向に床があり、床の上のセルと、キャラクターの上のセルとが空のときには、キャラクターを床の上に移動します。

ボタンを押したときには、進行方向のセルを調べます。セルが空のときには、新しく床を作ります。セルが床のときには、セルを空にすることによって、床を消します。

List 6-4 床を作って進む(CMakingFloorStageクラス)

```
// 移動処理
bool Move(const CInputState* is) {

    // 下のセルが空のときには、キャラクターは落下する
    if (Cell->Get(CX, CY+1)==' ') {
        CY++;
    } else

    // レバー入力に応じて、キャラクターを動かす
    if (!PrevLever) {

        // レバーを左に入力したときの処理
        if (is->Left) {

            // 左のセルが空ならば、キャラクターは左に進む
            if (Cell->Get(CX-1, CY)==' ') {
```

```
				CX--;
				VX=-1;
			} else

			// 左のセルが床で、左上と上のセルが空ならば、
			// キャラクターは床に飛び乗る
			if (
				Cell->Get(CX-1, CY)=='#' &&
				Cell->Get(CX-1, CY-1)==' ' &&
				Cell->Get(CX, CY-1)==' '
			) {
				CX--;
				CY--;
				VX=-1;
			}
		} else

		// レバーを右に入力したときの処理
		if (is->Right) {

			// 右のセルが空ならば、キャラクターは右に進む
			if (Cell->Get(CX+1, CY)==' ') {
				CX++;
				VX=1;
			} else

			// 右のセルが床で、右上と上のセルが空ならば、
			// キャラクターは床に飛び乗る
			if (
				Cell->Get(CX+1, CY)=='#' &&
				Cell->Get(CX+1, CY-1)==' ' &&
				Cell->Get(CX, CY-1)==' '
			) {
				CX++;
				CY--;
				VX=1;
			}
		}
	}
	PrevLever=is->Left||is->Right;

	// ボタンを押したときの処理
	if (!PrevButton && is->Button[0]) {

		// 進行方向のセルが空ならば、床を作る
		if (Cell->Get(CX+VX, CY)==' ') {
			Cell->Set(CX+VX, CY, '#');
		} else
```

```
        //進行方向のセルが床ならば、床を消して空にする
        if (Cell->Get(CX+VX, CY)=='#') {
            Cell->Set(CX+VX, CY, ' ');
        }
    }
    PrevButton=is->Button[0];

    return true;
}
```

SAMPLE

「MAKING FLOOR」は「床を作って進む」のサンプルです。

レバーの左右（カーソルキーの左右）でキャラクターが動きます。ボタン0（Zキー）を押すと、キャラクターの進行方向に床を作ります。床の前でボタン0を押すと、逆に床を壊すことができます。

床に向かって移動すると、キャラクターは床に飛び乗ります。床に乗ったり床を作ったりを繰り返すと、ステージの高いところまで登ることができます。

MAKING FLOOR → p. 390

サイコロを揃えて消す

同じ目のサイコロを上下左右に隣接させることによって、サイコロを消すアクションです。揃えなければならない個数が、サイコロの目に応じて変わります。

本書のサンプルでは、レバー入力でカーソルが上下左右に動きます（Fig. 6-24）。カーソルの

Fig. 6-24 ステージの構成

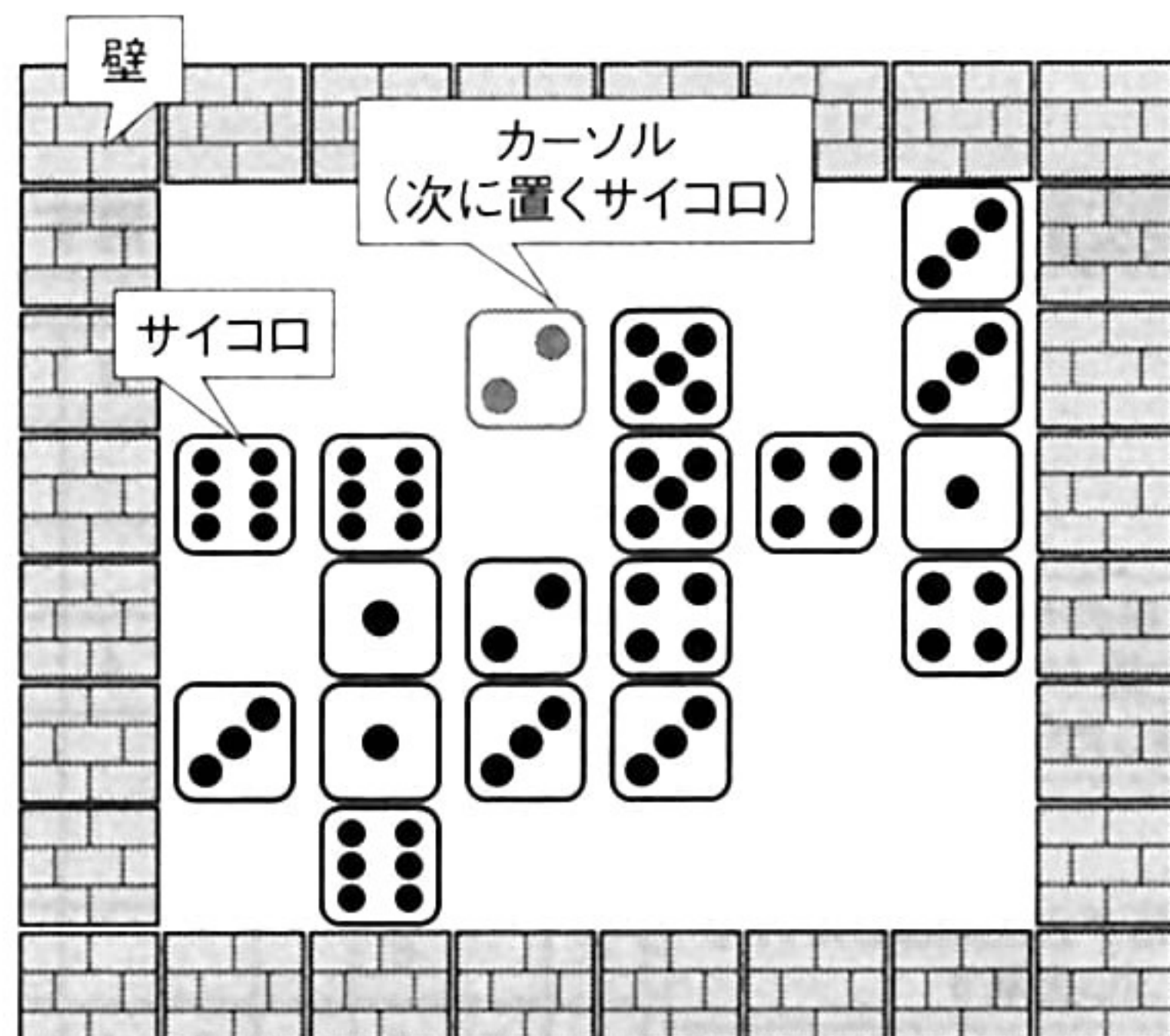
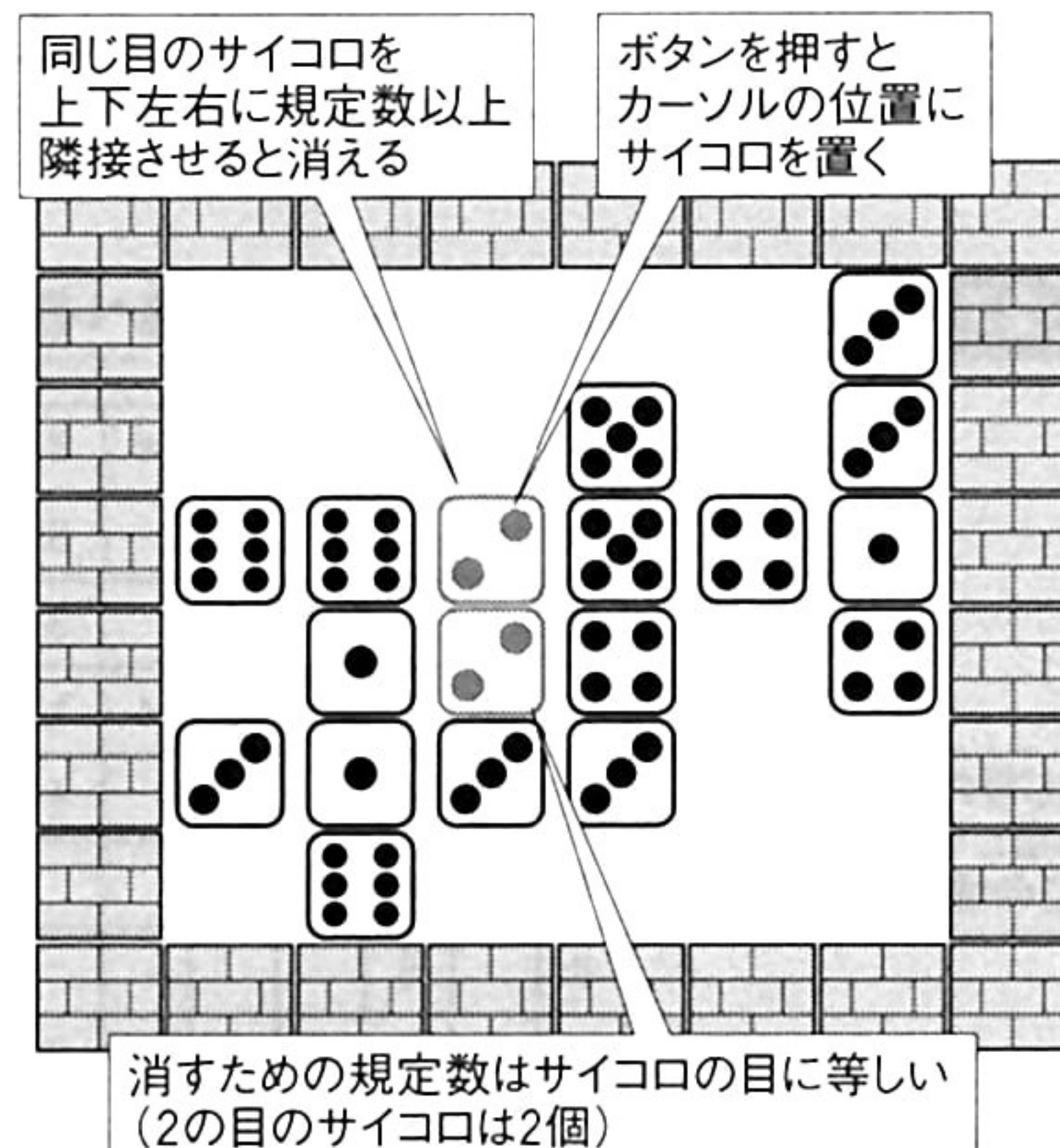



Fig. 6-25 サイコロを揃えて消す

位置には、次に置くサイコロが表示されています。

ボタンを押すと、カーソルの位置にサイコロを置くことができます。サイコロを置いたときに、同じ目のサイコロを上下左右に規定数以上隣接させると、サイコロが消えます（Fig. 6-25）。

規定数はサイコロの目と同じです。例えば2の目のサイコロは、2個以上を隣接させれば消すことができます。一方、6の目のサイコロは、6個以上を隣接させなければ消すことができません。

1の目のサイコロは単体では消えません。1の目のサイコロは、隣接する他の目のサイコロを消したときに、一緒に消すことができます。

サイコロを揃えて消すアクションは『XI』に採用されています。このゲームではサイコロを転がすことによって、サイコロの位置と目を変えることができます。サイコロの上面に出ている目の数だけ、同じ目のサイコロを隣接させると、消すことができます。サイコロを転がす方法は、キャラクターを操作してサイコロの上に登り、足で転がすという、ユニークな動きを採用しています。

アルゴリズム

サイコロを揃えて消すには、まずステージをセルで表現します（Fig. 6-26）。ステージの壁は「#」、サイコロは「1～6」の数字で表しました。数字はサイコロの目に対応しています。

新しくサイコロを置いたら、同じ目のサイコロが規定数以上隣接したかどうかを調べます。

Fig. 6-26 ステージをセルで表現する

#	#	#	#	#	#	#	#
#						3	#
#				5		3	#
#	6	6		5	4	1	#
#		1	2	4		4	#
#	3	1	3	3			#
#		6					#
#	#	#	#	#	#	#	#

壁（#）

サイコロ（1～6）

Fig. 6-27 隣接した同じ目のサイコロを調べる

#	#	#	#	#	#	#	#
#						3	#
#				5		3	#
#	6	6	2	5	4	1	#
#		1	2	4		4	#
#	3	1	3	3			#
#		6					#
#	#	#	#	#	#	#	#

同じ目のサイコロが隣接している数を調べる（この場合は2個）

Fig. 6-28 目の数以上隣接したサイコロを消す

#	#	#	#	#	#	#	#
#						3	#
#				5		3	#
#	6	6		5	4	1	#
#		1		4		4	#
#	3	1	3	3			#
#		6					#
#	#	#	#	#	#	#	#

同じ目のサイコロが規定数（サイコロの目）以上隣接したら消す

2の目が2個隣接したのでサイコロを消す

これは「連鎖的に消す」（→p. 106）や「ボールを入れ替える」（→p. 331）と同様の処理です。
サイコロの上下左右に、同じ目のサイコロが隣接しているかどうかを調べます（Fig. 6-27）。もしも隣接していたら、隣接したサイコロの上下左右も調べます。この手順を再帰的に行うことによって、隣接している同じ目のサイコロをすべて探し出します。
同じ目のサイコロが、目の数以上隣接していたら、サイコロを消します（Fig. 6-28）。隣接させる個数がサイコロの目によって変わることが、このアクションの特徴です。

プログラム

List 6-5はサイコロを揃えて消すプログラムです。ステージの移動処理、隣接した同じ目のサイコロを数える処理、隣接した同じ目のサイコロを消す処理を掲載しました。
移動処理（Move関数）は、初期状態・入力状態・消去判定状態・消去状態に分かれています。初期状態では、次に置くサイコロの目をランダムに決めて、入力状態に移行します。入力状態では、レバーの入力に応じて、カーソルを上下左右に動かします。ボタンを押したら、カーソルの位置に新しいサイコロを置き、消去判定状態に移行します。
消去判定状態では、ステージ内のすべてのサイコロについて、同じ目のサイコロが隣接する数を調べます。隣接する数がサイコロの目以上ならば、サイコロを消します。サイコロを消えるサイコロにして、消去状態に移行します。
消去状態では、一定時間が経過するのを待ってから、消えるサイコロを完全に消します。消えるサイコロは、時間とともにだんだん薄くなって消えるように表示します。具体的な処理はステージの描画処理（CDiceStageクラスのDraw関数）で行います。
隣接した同じ目のサイコロを数える処理（Count関数）では、上下左右に同じ目のサイコロが隣接しているかどうかを調べます。隣接していたら、隣接するセルの上下左右のセルについても再帰的に調べます。
隣接した同じ目のサイコロを消す処理（Erase関数）も同様に、上下左右に同じ目のサイコロが隣接しているかどうかを再帰的に調べて、消えるサイコロに変化させます。また、1の目のサイコロが消えるサイコロに隣接しているときにも、消えるサイコロに変えます。

List 6-5　サイコロを揃えて消す（CDiceStageクラス）

```
// 移動処理
bool CDiceStage::Move(const CInputState* is) {

	// セルの個数
	int xs=Cell->GetXSize(), ys=Cell->GetYSize();

	// 初期状態
	if (State==0) {

		// 次に置くサイコロの目をランダムに決める
```

```
        Type='1'+Rand.Int31()%DICE_TYPE;

        // 入力状態に移行する
        State=1;
    }

    // 入力状態
    if (State==1) {

        // レバーの入力に応じて、カーソルを上下左右に動かす
        if (!PrevLever) {
            int cx=CX, cy=CY;
            if (is->Left) cx--; else
            if (is->Right) cx++; else
            if (is->Up) cy--; else
            if (is->Down) cy++;

            // 移動先が壁でなければ、カーソルを移動する
            if (Cell->Get(cx, cy)!='#') {
                CX=cx;
                CY=cy;
            }
        }
        PrevLever=is->Left||is->Right||is->Up||is->Down;

        // 空のセルでボタンを押したら、その場所にサイコロを置く
        if (!PrevButton && is->Button[0] && Cell->Get(CX, CY)==' ') {

            // サイコロの目をセルに書き込む
            Cell->Set(CX, CY, Type);

            // 消去判定状態に移行する
            State=2;
        }
        PrevButton=is->Button[0];
    }

    // 消去判定状態
    if (State==2) {

        // サイコロが消えなかったら、初期状態に移行する
        State=0;

        // ステージ内のすべてのサイコロについて調べる
        for (int y=0; y<ys; y++) {
            for (int x=0; x<xs; x++) {

                // 同じ目のサイコロが規定数以上隣接していたら、
                // サイコロを消す
```

```
                char c=Cell->Get(x, y);
                if (
                    '1'<c && c<'1'+DICE_TYPE &&
                    Count(x, y, c)>=c-'0'
                ) {
                    // サイコロを消えるサイコロに変化させる
                    Erase(x, y, c|0x80);

                    // タイマーを設定して、消去状態に移行する
                    Time=0;
                    State=3;
                }
            }
        }

        // サイコロのカウントずみのマークを解除する
        for (int y=0; y<ys; y++) {
            for (int x=0; x<xs; x++) {
                Cell->Set(x, y, Cell->Get(x, y)&0x7f);
            }
        }
    }

    // 消去状態
    if (State==3) {

        // 一定時間が経過するのを待ってから、
        // 消えるサイコロを完全に消去する
        Time++;
        if (Time==30) {
            for (int x=0; x<xs; x++) {
                for (int y=0; y<ys; y++) {

                    // 消えるサイコロを見つけたら、
                    // セルを空にする
                    if (Cell->Get(x, y)&0x40) {
                        Cell->Set(x, y, ' ');
                    }
                }
            }

            //  初期状態に移行する
            State=0;
        }
    }

    return true;
}
```

```
//隣接した同じ目のサイコロを数える処理
int CDiceStage::Count(int x, int y, char c) {

    //現在のセルが指定された種類かどうかを調べる
    if (Cell->Get(x, y)==c) {

        //指定された種類ならば、
        //セルにカウントずみのマークを付ける
        //（最上位ビットを1にする）
        Cell->Set(x, y, c|0x80);

        //上下左右に隣接するセルについても再帰的に調べて、
        //同じ種類のセルが隣接する個数の合計を返す
        return
            1+
            Count(x-1, y, c)+
            Count(x+1, y, c)+
            Count(x, y-1, c)+
            Count(x, y+1, c);
    }

    //現在のセルが指定された種類でなければ、
    //同じ種類のセルが隣接する個数として0を返す
    return 0;
}

//隣接した同じ目のサイコロを消す処理
void CDiceStage::Erase(int x, int y, char c) {

    //現在のセルが指定された種類かどうかを調べる
    char d=Cell->Get(x, y);
    if (d==c) {

        //指定された種類ならば、消えるサイコロにする
        //（上から2番目のビットを1にする）
        Cell->Set(x, y, d|0x40);

        //上下左右に隣接するセルについても再帰的に調べて、
        //同じ種類のセルを消えるサイコロにする
        Erase(x-1, y, c);
        Erase(x+1, y, c);
        Erase(x, y-1, c);
        Erase(x, y+1, c);
    } else

    //現在のセルが1の目のサイコロならば、
    //消えるサイコロにする
    //（上下左右に隣接するセルについては調べない）
    if (d=='1') {
```

```
            Cell->Set(x, y, d|0x40);
        }
    }
```

SAMPLE

「DICE」は「サイコロを揃えて消す」のサンプルです。

レバーの上下左右(カーソルキーの上下左右)でカーソルが動きます。カーソルの位置には、次に置くサイコロが表示されています。ボタン0を押すと、その場所にサイコロを置きます。

サイコロを置いたときに、同じ目のサイコロが規定数以上隣接すると、サイコロを消すことができます。規定数はサイコロの目と同じです。例えば、2の目のサイコロは2個以上、6の目のサイコロは6個以上を、隣接させる必要があります。

1の目のサイコロは、単体では消えません。他の目のサイコロが消えるときに、1の目のサイコロを隣接させておくと、消すことができます。

DICE → p. 390

建物を建てる

建物を特定の配置に並べることによって、だんだん大きな建物を建てていくアクションです。小さな建物で空き地を囲むことによって、空き地により大きな建物を建てることができます。周囲の状況によって、自動的に新しい建物が発生する様子は、ライフゲームに似ています。

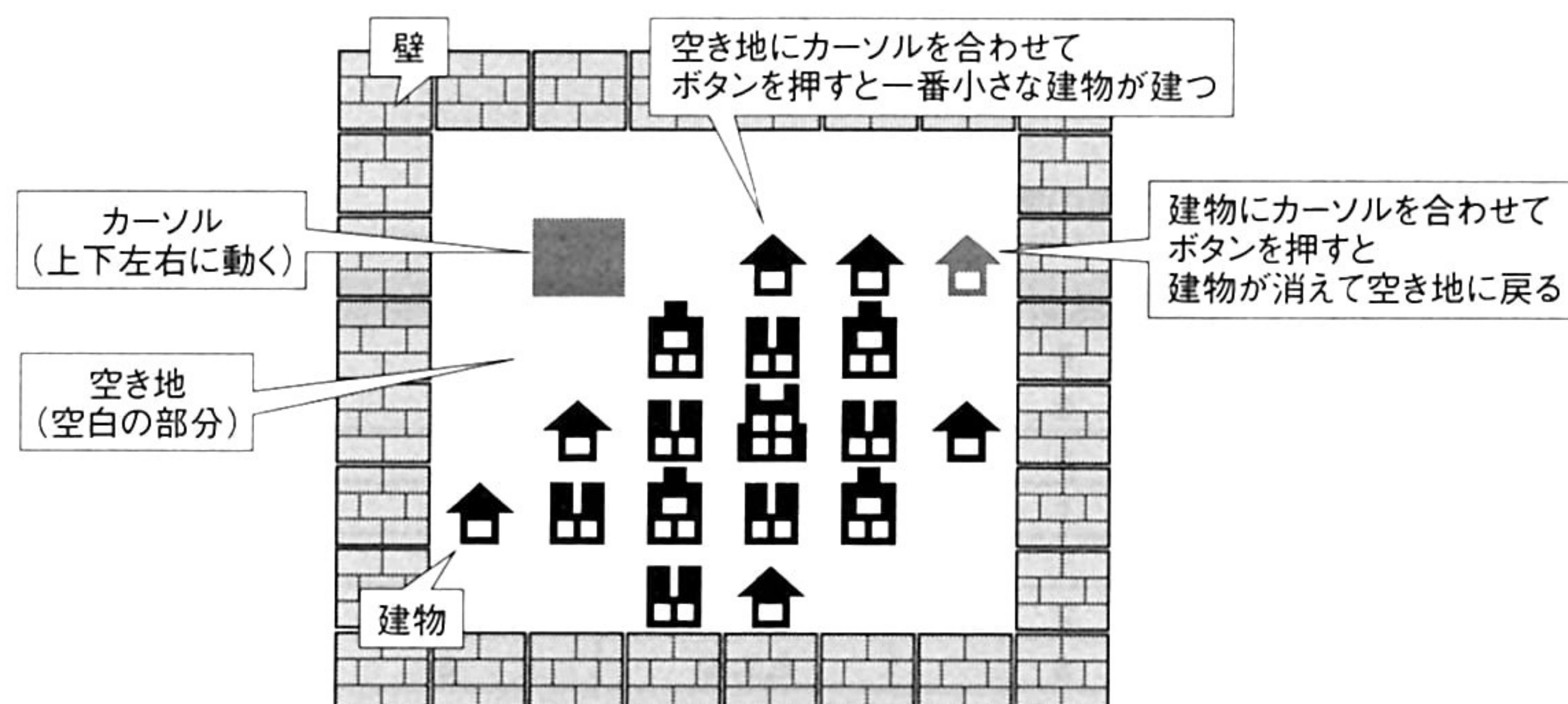
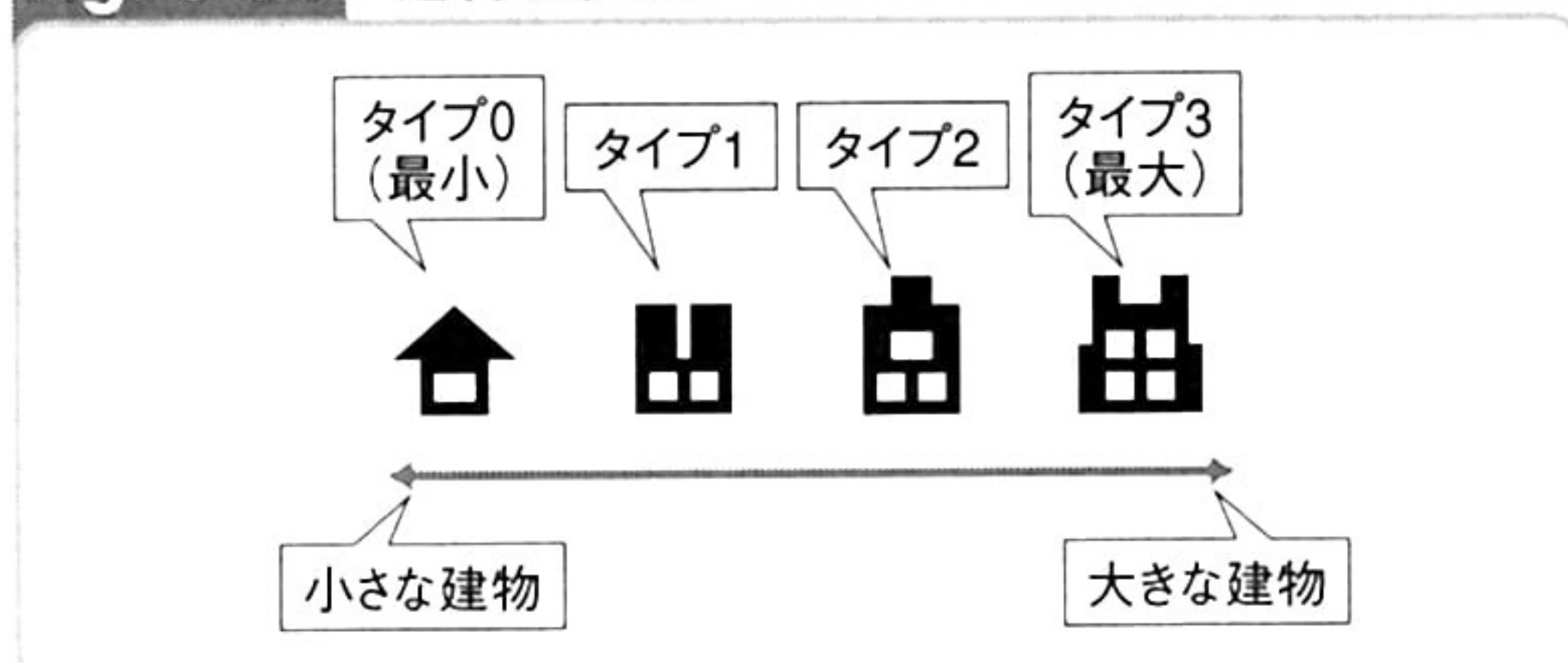
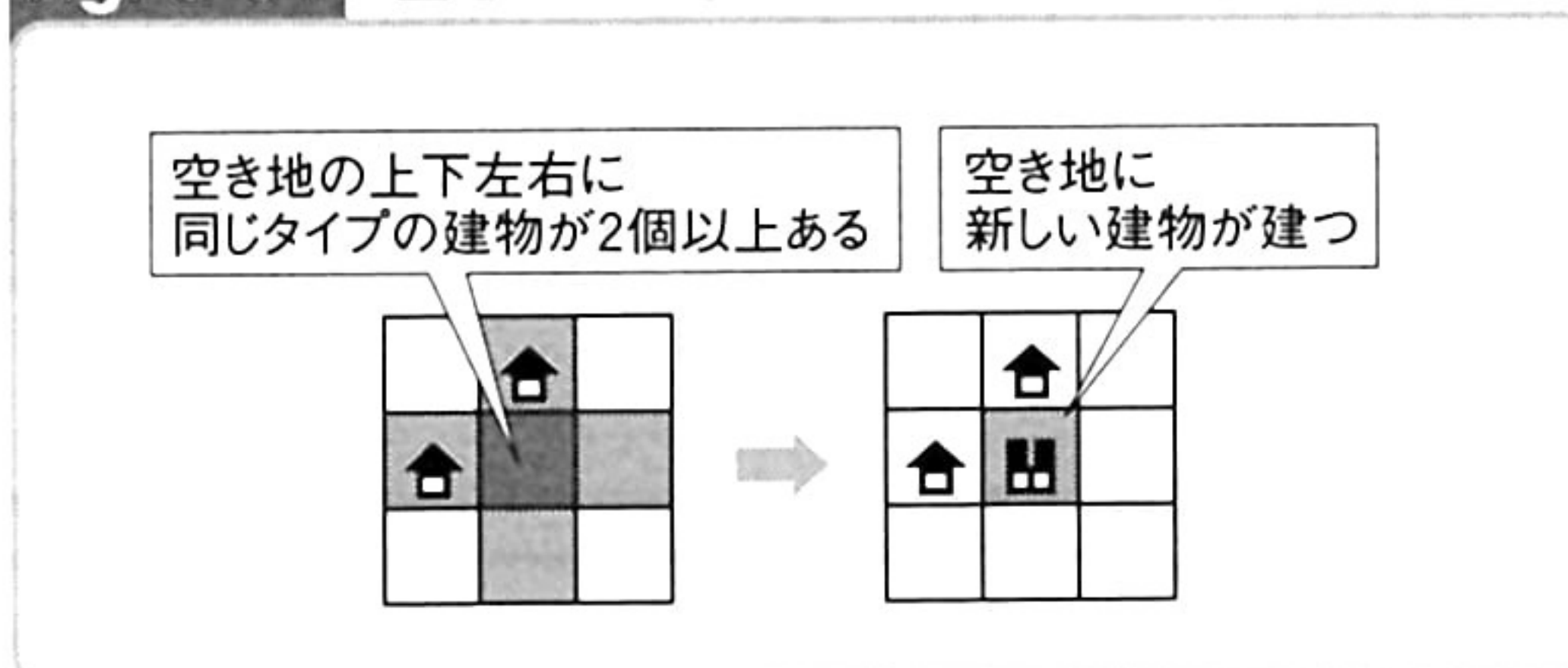
本書のサンプルでは、レバー操作でカーソルが上下左右に動きます(Fig. 6-29)。ボタンを押すと、カーソルの位置に一番小さな建物を建てることができます。建物の上でボタンを押すと、建物が消えて、空き地に戻ります。

建物には4つの種類があります(Fig. 6-30)。建物の規模が大きくなるほど、窓を多くしまし

Fig. 6-29 ステージの構成

Fig. 6-30 建物の種類

タイプ0（最小）
タイプ1
タイプ2
タイプ3（最大）
小さな建物
大きな建物

Fig. 6-31 空き地に建物が建つ

空き地の上下左右に同じタイプの建物が2個以上ある
空き地に新しい建物が建つ

た。これらの建物を、タイプ0〜3と呼ぶことにします。

空き地の上下左右に同じタイプの建物が2個以上あると、空き地に新しい建物が建ちます(Fig. 6-31)。新しい建物のタイプは、周りの建物のタイプと数に応じて決まります。

例えば、タイプ0の建物が2個あるときには、タイプ1の建物が建ちます(Fig. 6-32)。タイプ1の建物が2個あると、タイプ2の建物が建ちます。このように、2個の建物があると、1つ上のタイプの建物が建ちます。

一方、タイプ0の建物が3個あると、タイプ2の建物が建ちます(Fig. 6-33)。タイプ0の建物が4個あると、タイプ3の建物が建ちます。このように、3個または4個の建物があると、2つまたは3つ上のタイプの建物が建ちます。

本書のサンプルでは、タイプ3よりも大きな建物はありません。Fig. 6-345は、周囲の建物と、新しく建つ建物のタイプの関係をまとめたものです。

Fig. 6-32 2個の建物がある場合

タイプ0の建物が2個ある
タイプ1の建物が建つ
タイプ1の建物が2個ある
タイプ2の建物が建つ

Fig. 6-33 3個以上の建物がある場合

タイプ0の建物が3個ある
タイプ2の建物が建つ
タイプ0の建物が4個ある
タイプ3の建物が建つ

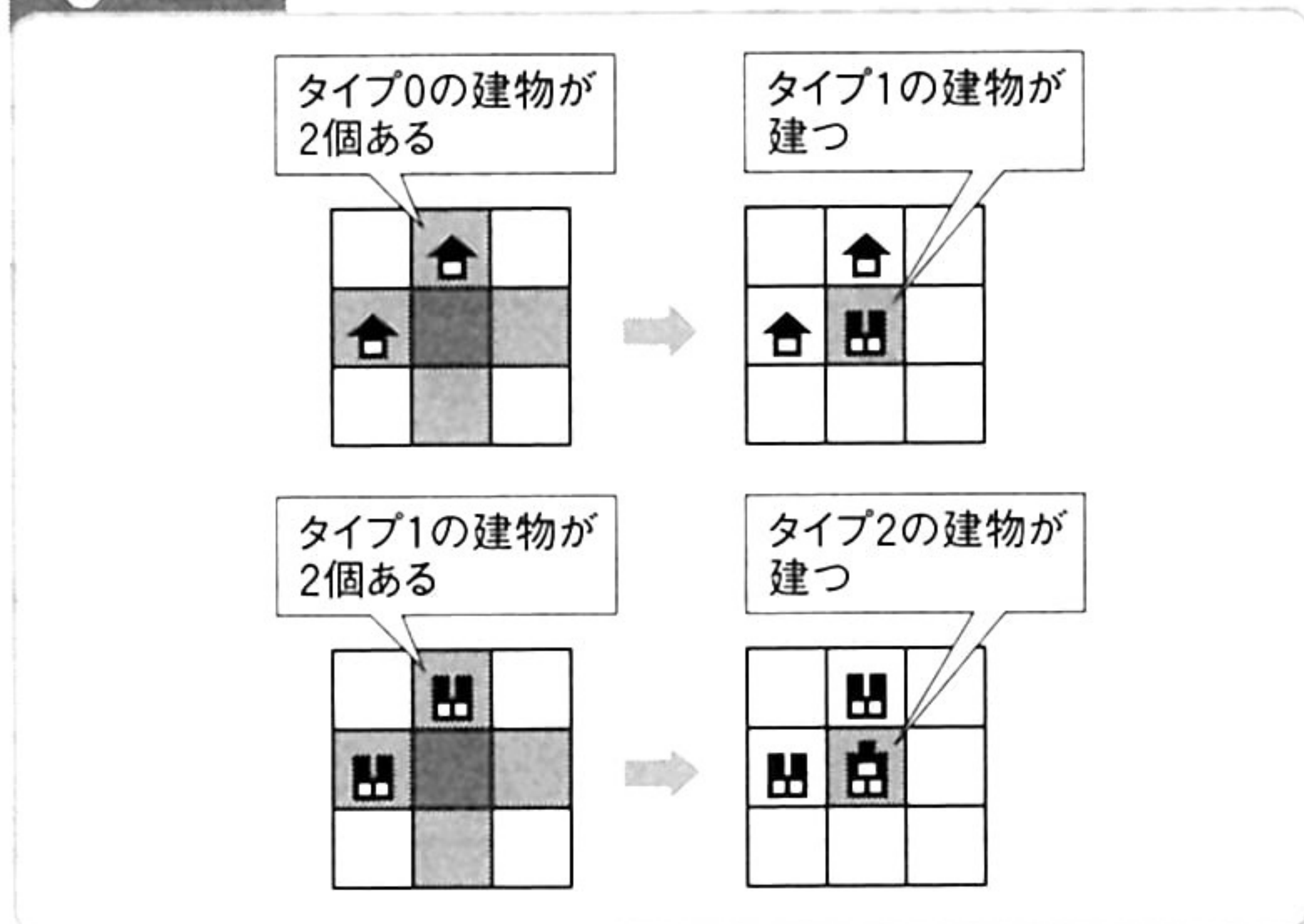
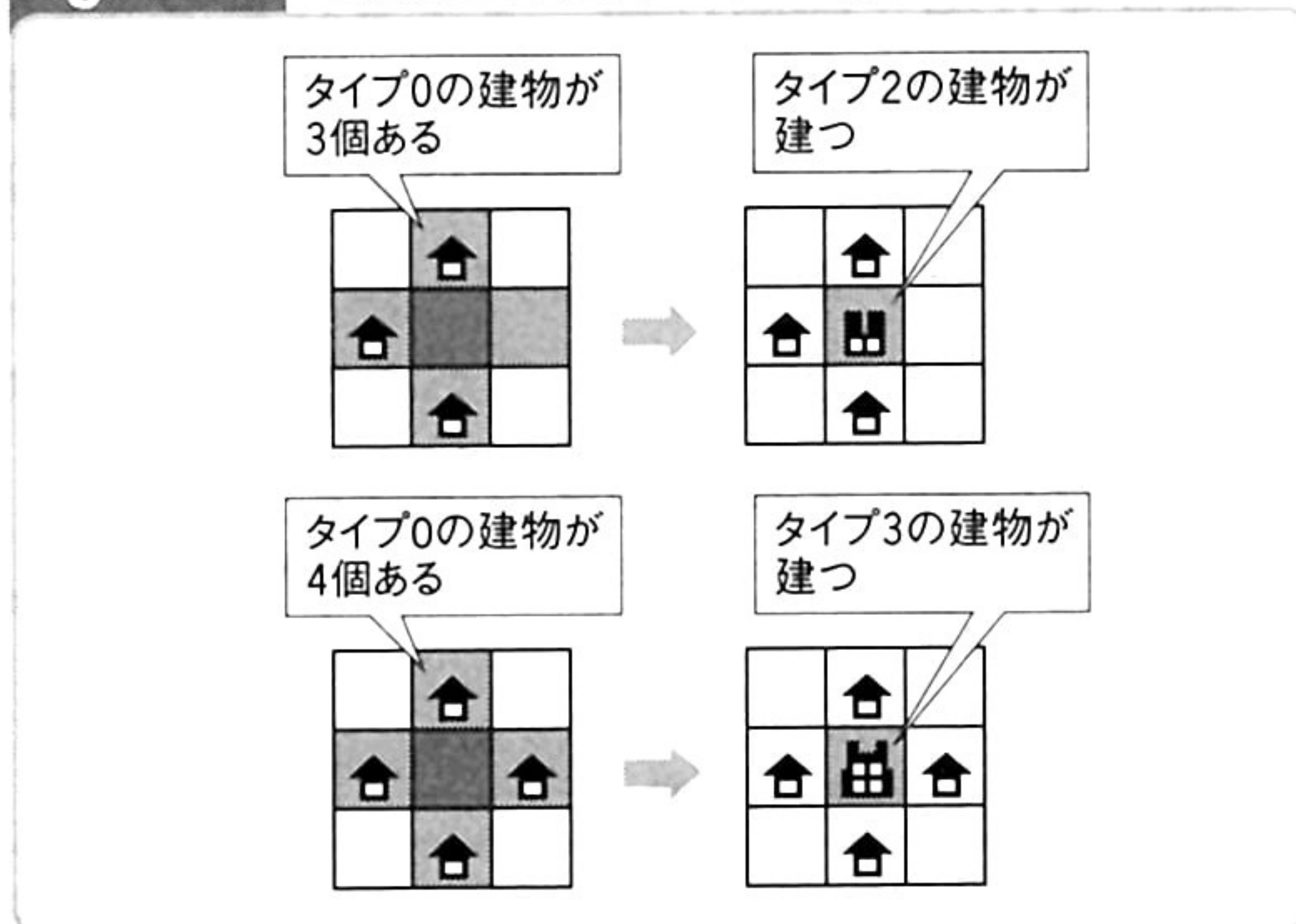



Fig. 6-34 周囲の建物と新しく立つ建物の関係

周囲の建物のタイプ	周囲の建物の数	建つ建物のタイプ
0	2	1
0	3	2
0	4	3
1	2	2
1	3、4	3
2	2、3、4	3
3	2、3、4	3

建物を建てるアクションは『オトスタツ』に採用されています。このゲームでは、ステージに「土」「水」「木」を置きます。これらを特定の配置にすると、建物が建ちます。

同じ大きさの2個以上の建物で囲んだ場所には、新しい建物が建ちます。大きな建物で囲むほど、また多くの建物で囲むほど、新しい建物は大きくなります。大きな建物を数多く建てることが、ゲームの目的です。

建物は立体的な折り紙細工のように表現されているので、見た目にも楽しめるゲームです。建物が増えると、ステージはまるで大都会のようになります。

アルゴリズム

建物を建てるアクションを実現するには、まずステージをセルで表現します（Fig. 6-35）。ステージの壁は「#」、建物は「0〜3」の数字で表しました。数字は建物のタイプに対応しています。

ボタンを押したら、カーソルの位置にタイプ0の建物を建てます。そして、ステージ内のすべての空き地について、周囲に建っている建物を調べます。

タイプ0から3まで順番に、2個以上の同じタイプの建物が、空き地の上下左右にあるかどうかを調べます（Fig. 6-36）。2個以上ある場合には、空き地に新しい建物を建てます（Fig. 6-37）。

新しい建物のタイプは、

周囲の建物のタイプ+周囲の建物の数-1

Fig. 6-35 ステージをセルで表現する

#	#	#	#	#	#	#	#
#							#
#				0	0		#
#			2	1	2		#
#		0	1	3	1	0	#
#	0	1	2	1	2		#
#			1	0			#
#	#	#	#	#	#	#	#

壁（#）

建物（0〜3）

Fig. 6-36 空き地の周囲にある同じタイプの建物を数える

#	#	#	#	#	#	#	#
#							#
#							#
#				0			#
#			0				#
#							#
#							#
#	#	#	#	#	#	#	#

空き地の周囲にある同じタイプの建物を数える

Fig. 6-37 空き地に新しい建物を建てる

#	#	#	#	#	#	#	#
#							#
#							#
#			1	0			#
#			0				#
#							#
#							#
#	#	#	#	#	#	#	#

同じタイプの建物が2個以上あったら空き地に建物を建てる

のように計算します。
例えば、タイプ0の建物が2個あるときには、

0+2-1=1

となるので、タイプ1の建物を建てます。また、タイプ1の建物が3個あるときには、

1+3-1=3

となるので、タイプ3の建物を建てます。最大の建物はタイプ3なので、計算結果が3を超えるときには、タイプ3の建物を建てます。

プログラム

List 6-6は建物を建てるプログラムです。ステージの移動処理を掲載しました。
移動処理では、レバー入力に応じてカーソルを上下左右に動かします。ボタンを押したら、カーソルの位置にタイプ0の建物を建てます。カーソルの位置に建物があるときには、建物を消して、空のセルにします。
次に、ステージ内のすべての空き地について、周囲のセルを調べます。同じタイプの建物が2個以上あったら、空き地に新しい建物を建てます。新しい建物のタイプは、前述の計算方法で決めます。
異なるタイプの建物が、それぞれ2個ずつあるときには、より大きなタイプの建物を建てます。例えば、タイプ0の建物が2個、タイプ1の建物が2個あるときには、タイプ2の建物を建てます。

List 6-6 建物を建てる(CBuildingStageクラス)

```
// 移動処理
bool Move(const CInputState* is) {

    // セルの個数
    int xs=Cell->GetXSize(), ys=Cell->GetYSize();

    // レバー入力に応じて、カーソルを上下左右に動かす
    if (!PrevLever) {
        int cx=CX, cy=CY;
        if (is->Left) cx--;
        if (is->Right) cx++;
        if (is->Up) cy--;
        if (is->Down) cy++;

        // 移動先が壁でなければ、カーソルを動かす
        if (Cell->Get(cx, cy)!='#') {
```

```
            CX=cx;
            CY=cy;
        }
    }
    PrevLever=is->Left||is->Right||is->Up||is->Down;

    // ボタンを押したときの処理
    if (!PrevButton && is->Button[0]) {

        // カーソルの位置にあるセルが空ならば、
        // タイプ0の建物を建てる
        if (Cell->Get(CX, CY)==' ') {
            Cell->Set(CX, CY, '0');
        } else

        // カーソルの位置にあるセルが空でなければ、
        // 建物を消して空にする
        {
            Cell->Set(CX, CY, ' ');
        }
    }
    PrevButton=is->Button[0];

    // 同じタイプの建物に囲まれた空き地を探す
    for (int y=0; y<ys; y++) {
        for (int x=0; x<xs; x++) {

            // 空き地を見つけたときの処理
            if (Cell->Get(x, y)==' ') {

                // 上下左右にあるセルの相対位置
                static const int
                    vx[]={-1, 1, 0, 0},
                    vy[]={0, 0, -1, 1};

                // 新しく建つ建物のタイプ
                int type=0;

                // タイプ0～3の建物が、
                // 空き地の周りにそれぞれ何個あるのかを調べる
                for (int i=0; i<BUILDING_TYPE; i++) {

                    // 指定されたタイプの建物を数える
                    int count=0;
                    for (int j=0; j<4; j++) {
                        if (Cell->Get(x+vx[j], y+vy[j])=='0'+i) {
                            count++;
                        }
                    }
```

```
                // 同じタイプの建物が2個以上あるときには、
                // 新しく建つ建物のタイプを計算する
                if (count>=2) {
                    type=i+count-1;
                }
            }

            // 新しく建物が建つ場合には、
            // タイプ3を上限として、建物を建てる
            if (type>0) {
                Cell->Set(
                    x, y,
                    '0'+min(type, BUILDING_TYPE-1));
            }
        }
    }
}

return true;
}
```

SAMPLE

「BUILDING」は「建物を建てる」のサンプルです。

レバーの上下左右(カーソルキーの上下左右)でカーソルが動きます。ボタン0(Zキー)を押すと、カーソルの位置にタイプ0の建物(一番小さな建物)を建てることができます。すでに建物が建っている場所の上でボタン0を押すと、空き地にすることができます。

同じタイプの建物で囲まれた空き地には、新しい建物が建ちます。同じタイプの建物が2個あるときには、1つ大きなタイプの建物が建ちます。3個または4個あるときには、2つまたは3つ大きなタイプの建物が建ちます。ただし、タイプ3よりも大きな建物は建ちません。

例えば、タイプ0の建物が2個あると、タイプ1の建物が建ちます。タイプ1の建物が3個あると、タイプ3の建物が建ちます。

BUILDING → p. 390

まとめ

本章では、これまでの章では紹介しなかった、少し変わったアクションを紹介しました。パズルゲームには無数の種類があるように思えますが、多くのゲームは他のゲームのルールを踏襲していたり、いくつかのゲームのルールを組み合わせていたりして、まったくオリジナルのゲームというのは意外にありません。その点、ルールが個性的なゲームには、グラフィックやサウンドも一風変わっていて、他にはない魅力を持った作品が多いように思えます。

というわけで、「定番のルールにとらわれず、今までになかったゲームを考え出そう!」というのが本章のまとめです。

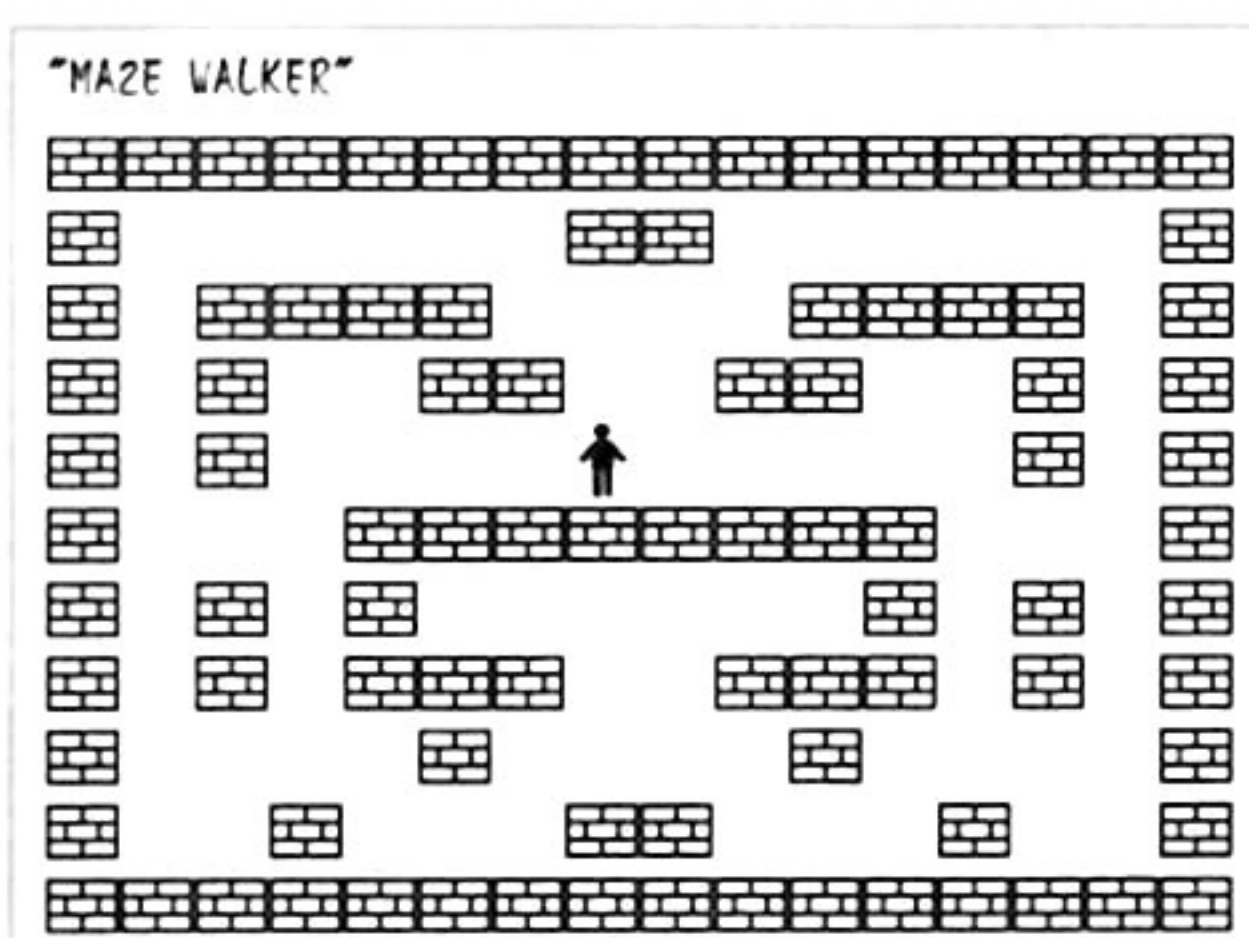
Bonus Stage

付録

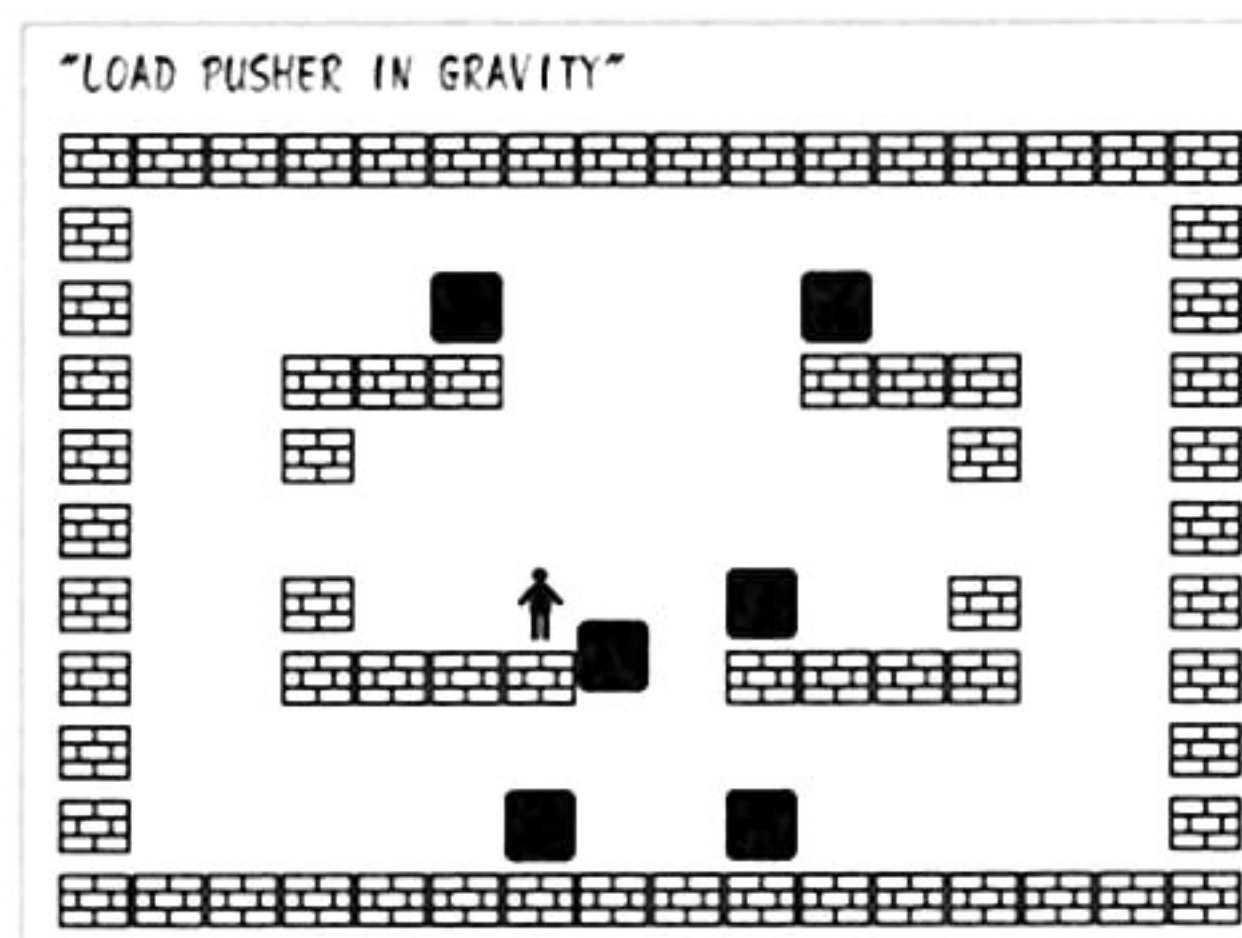
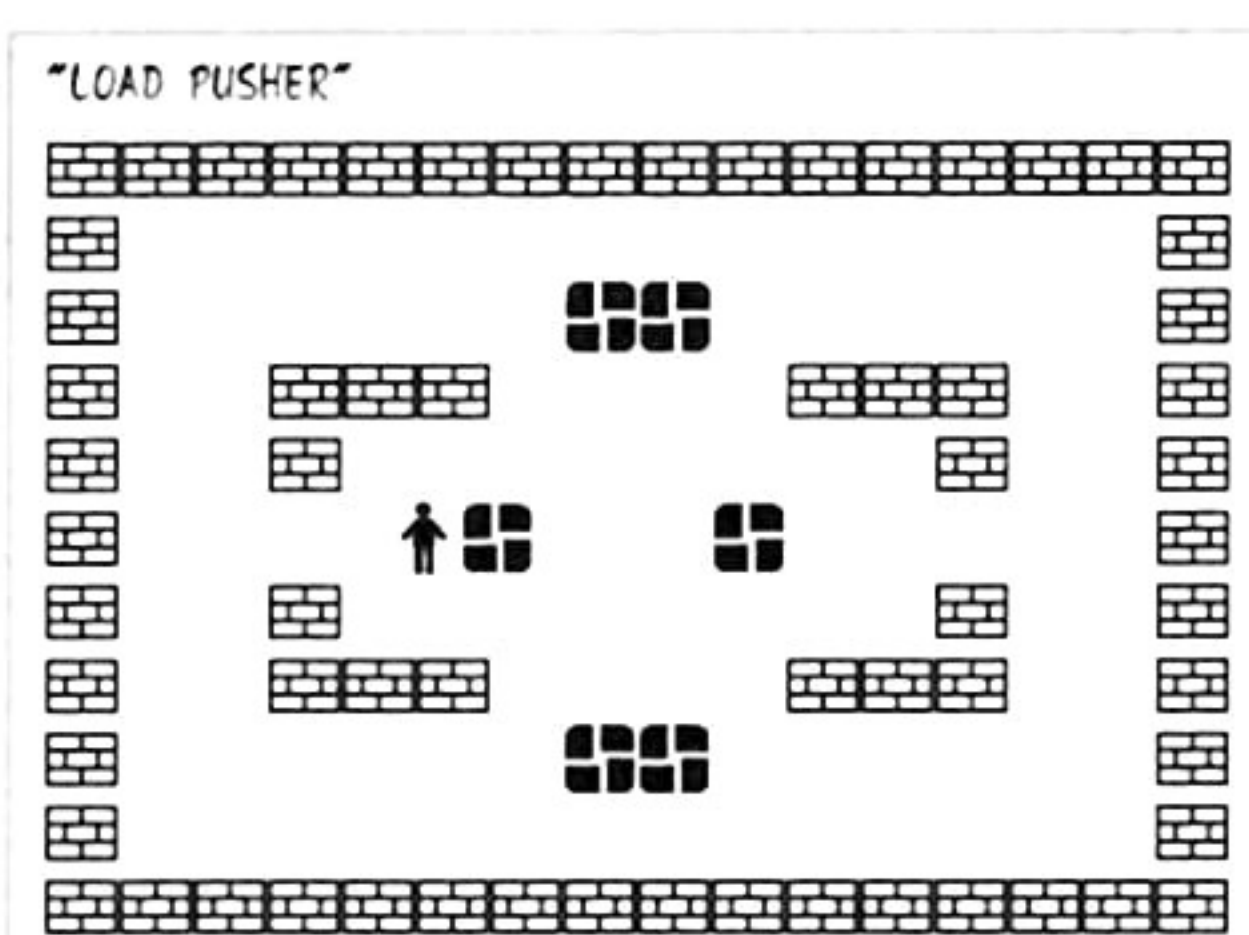
Appendix

デモプログラム一覧
引用ゲーム一覧
索引

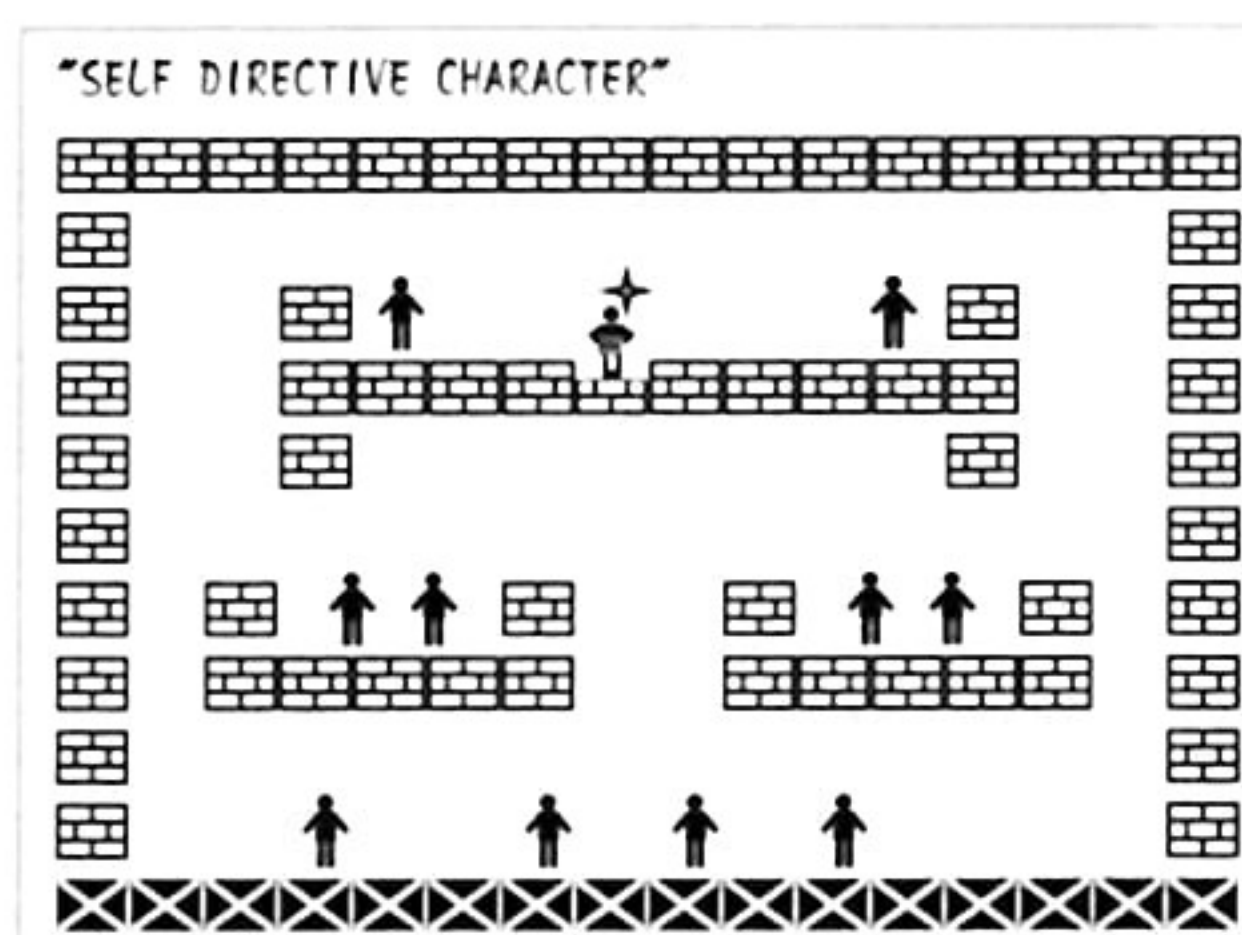
デモプログラム一覧　Appendix 01

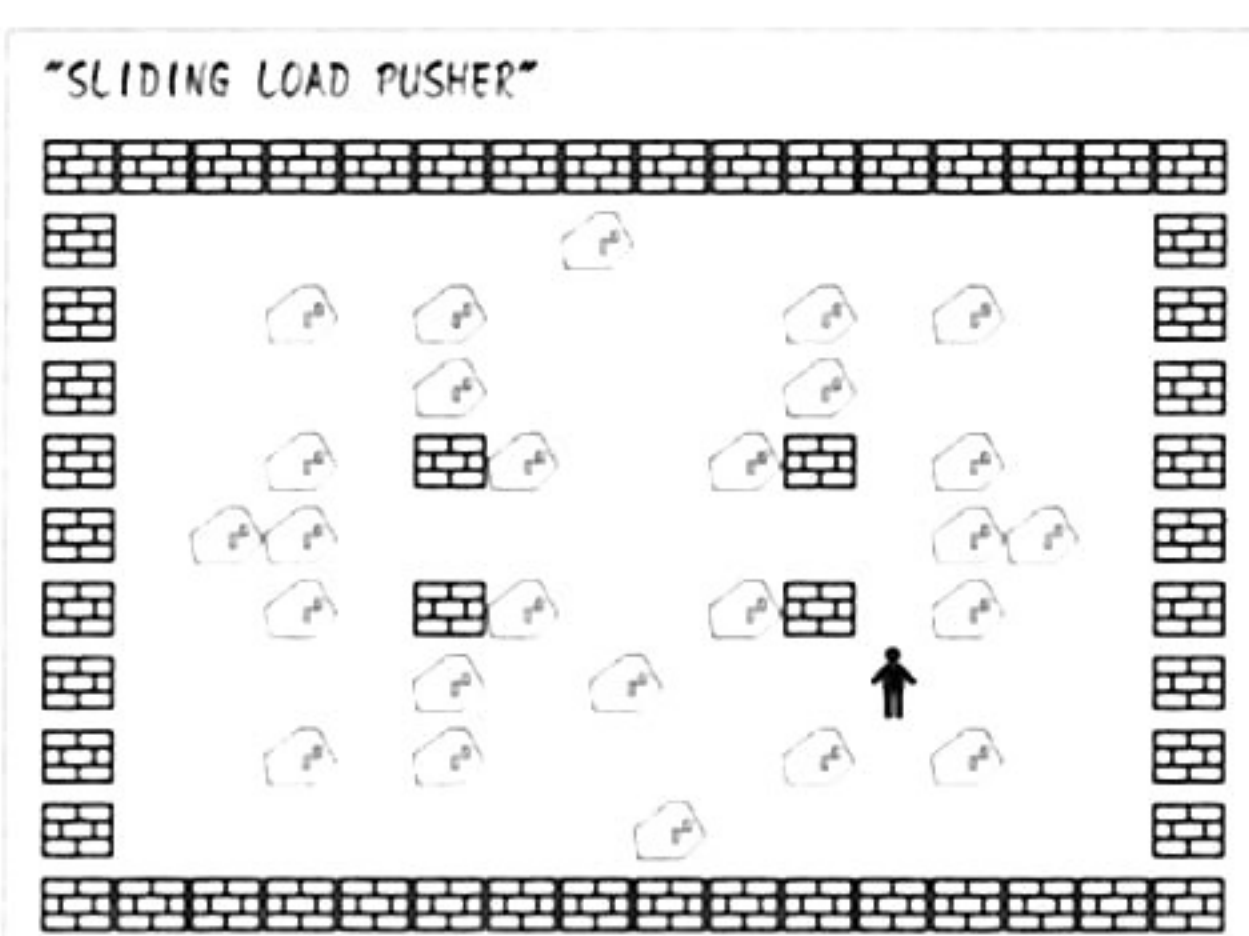
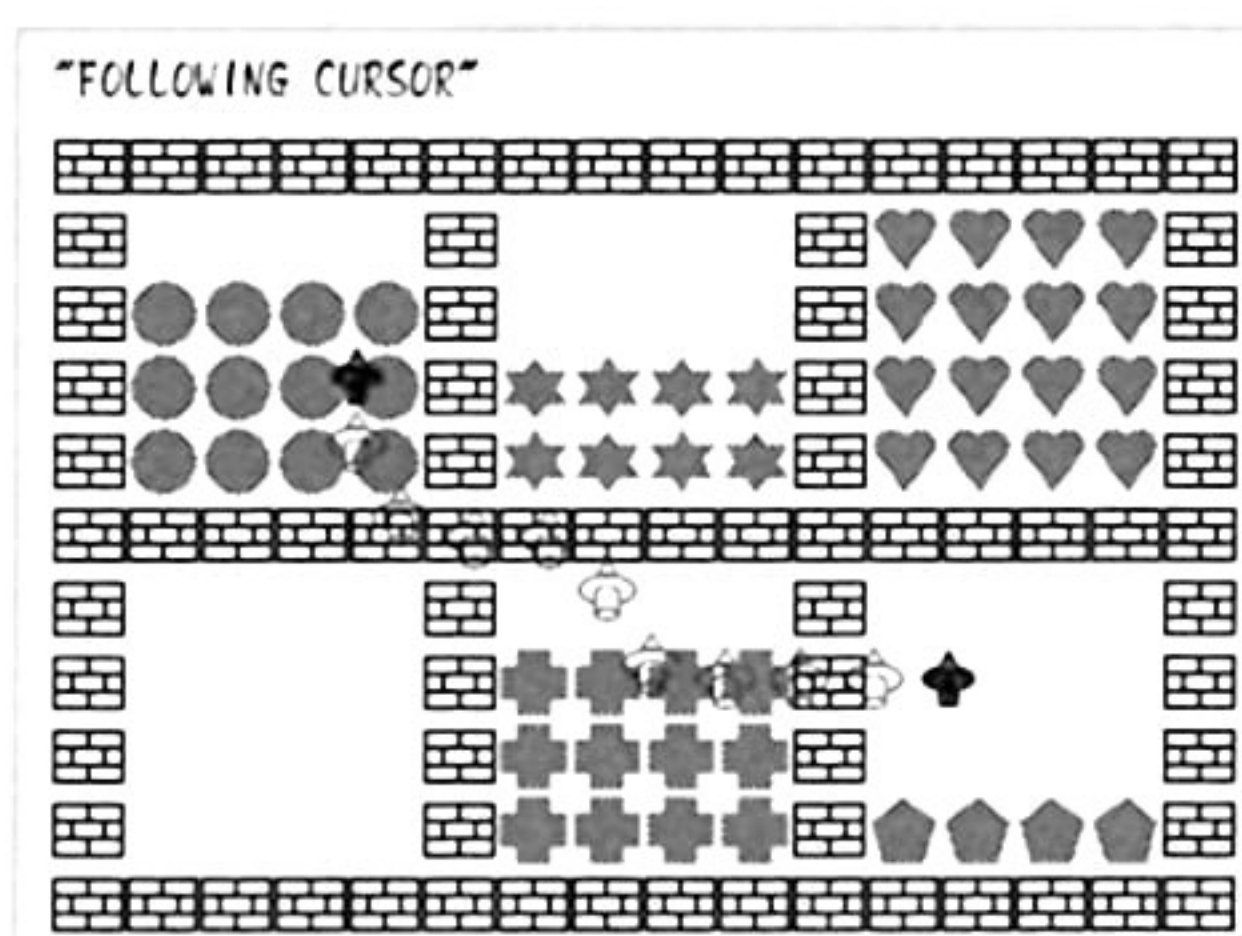
Stage01　動かす　Move

"MAZE WALKER"

MAZE WALKER
→ p. 10
「迷路を歩く」
↑↓←→：キャラクターの移動

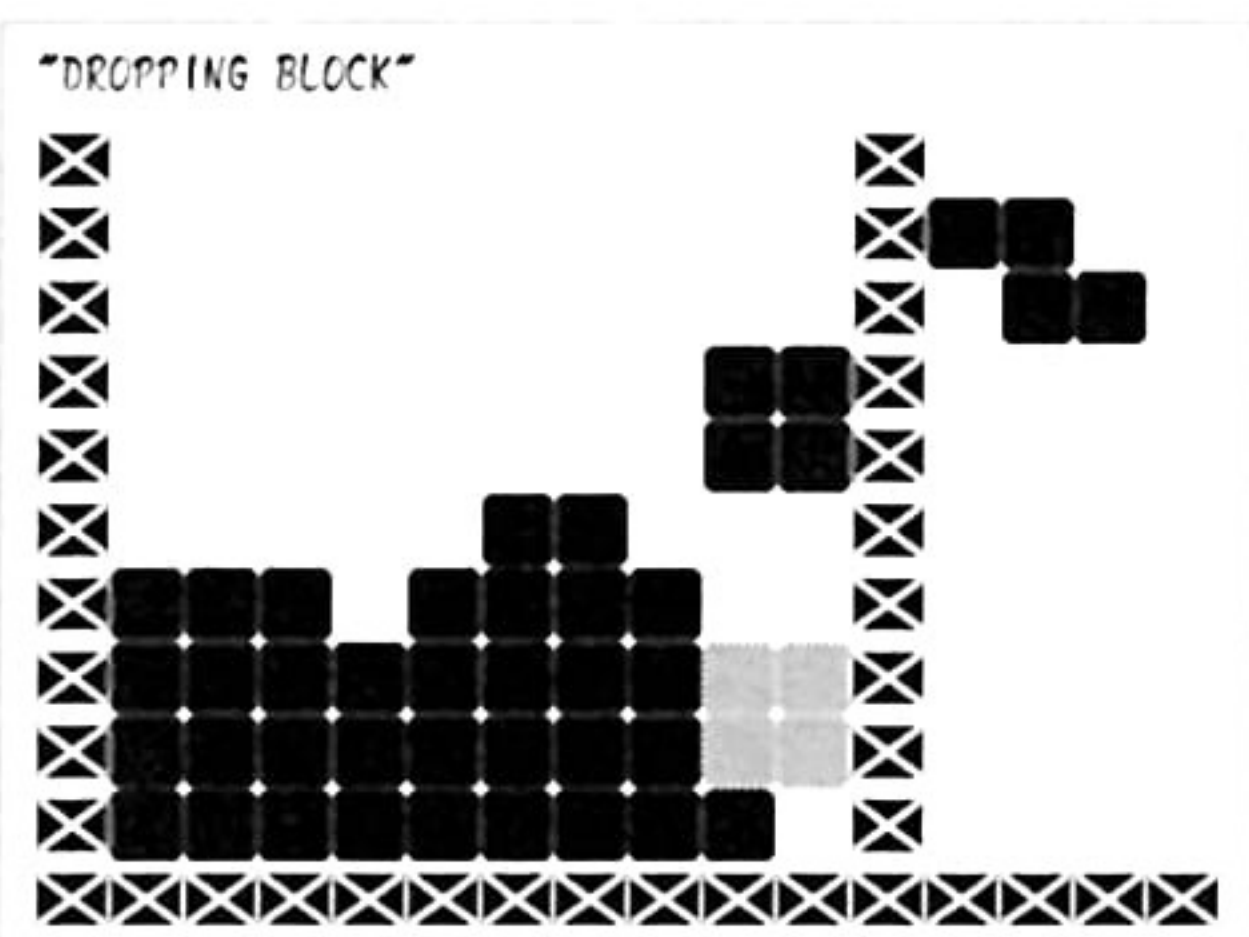
"LOAD PUSHER IN GRAVITY"

LOAD PUSHER IN GRAVITY
→ p. 30
「重力で落ちる荷物を押す」
↑↓←→：キャラクターの移動

"LOAD PUSHER"

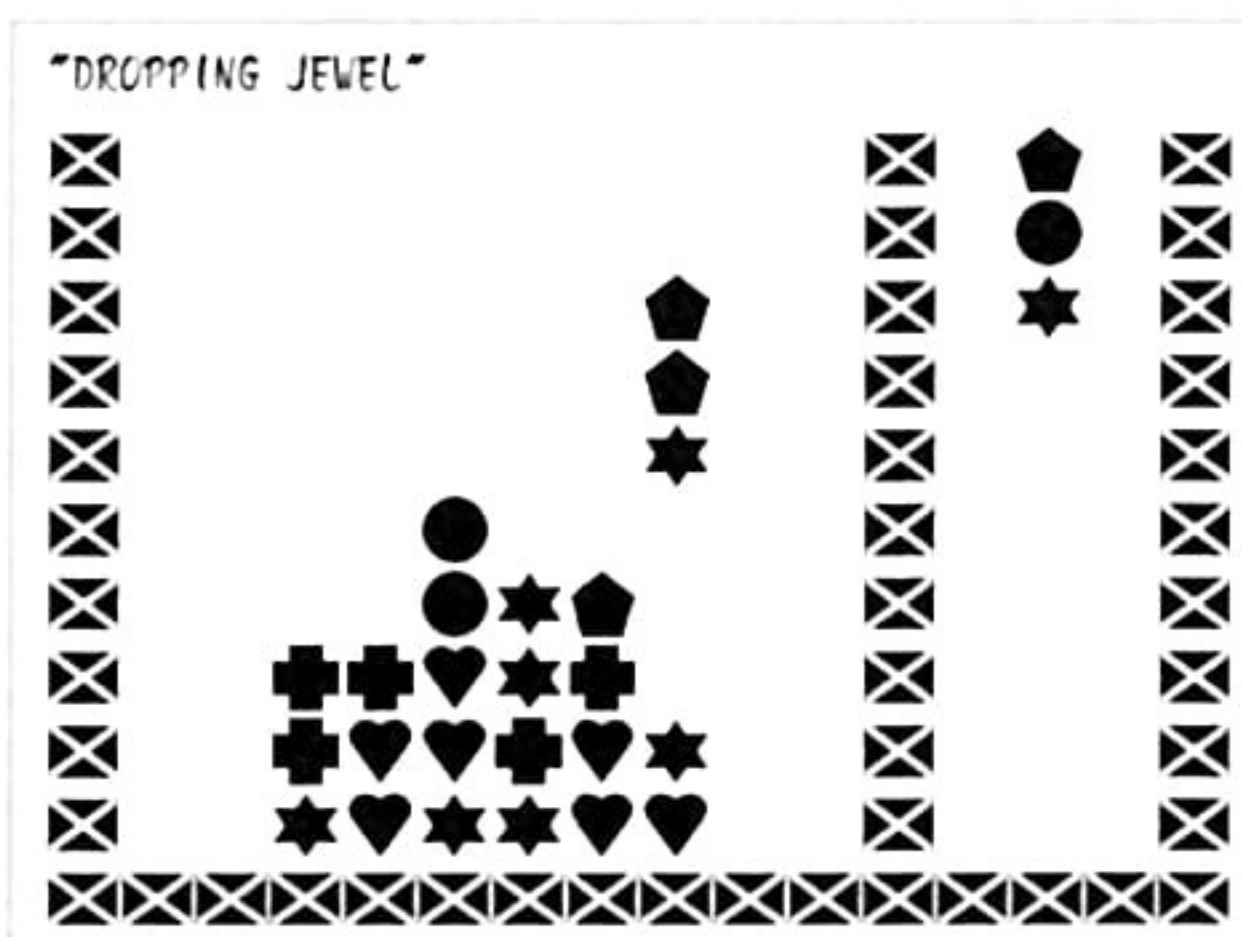
LOAD PUSHER
→ p. 19
「荷物を押す」
↑↓←→：キャラクターの移動

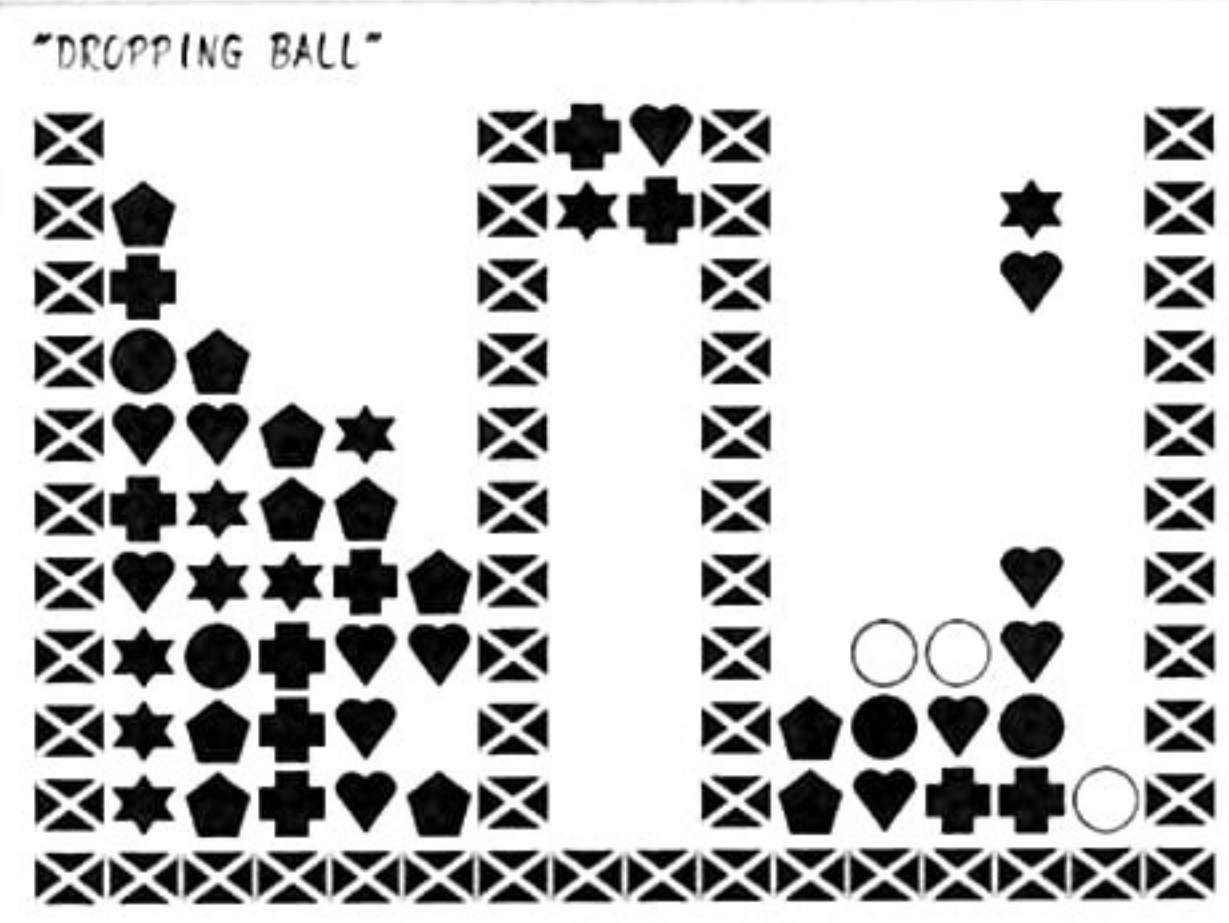
"SELF DIRECTIVE CHARACTER"

SELF DIRECTIVE CHARACTER
→ p. 36
「自律的に動くキャラクター」
ボタン0（Zキー）：キャラクターに命令する

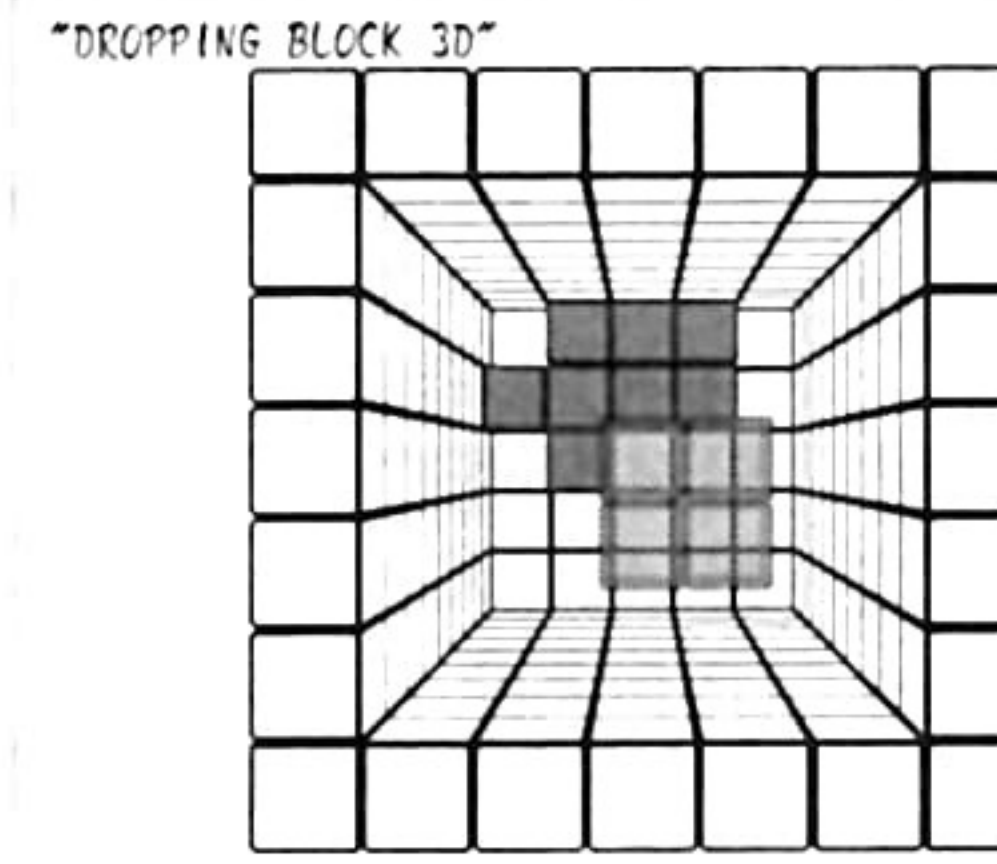
"SLIDING LOAD PUSHER"

SLIDING LOAD PUSHER
→ p. 25
「滑る荷物を押す」
↑↓←→：キャラクターの移動
ボタン0（Zキー）：荷物を動かす

"FOLLOWING CURSOR"

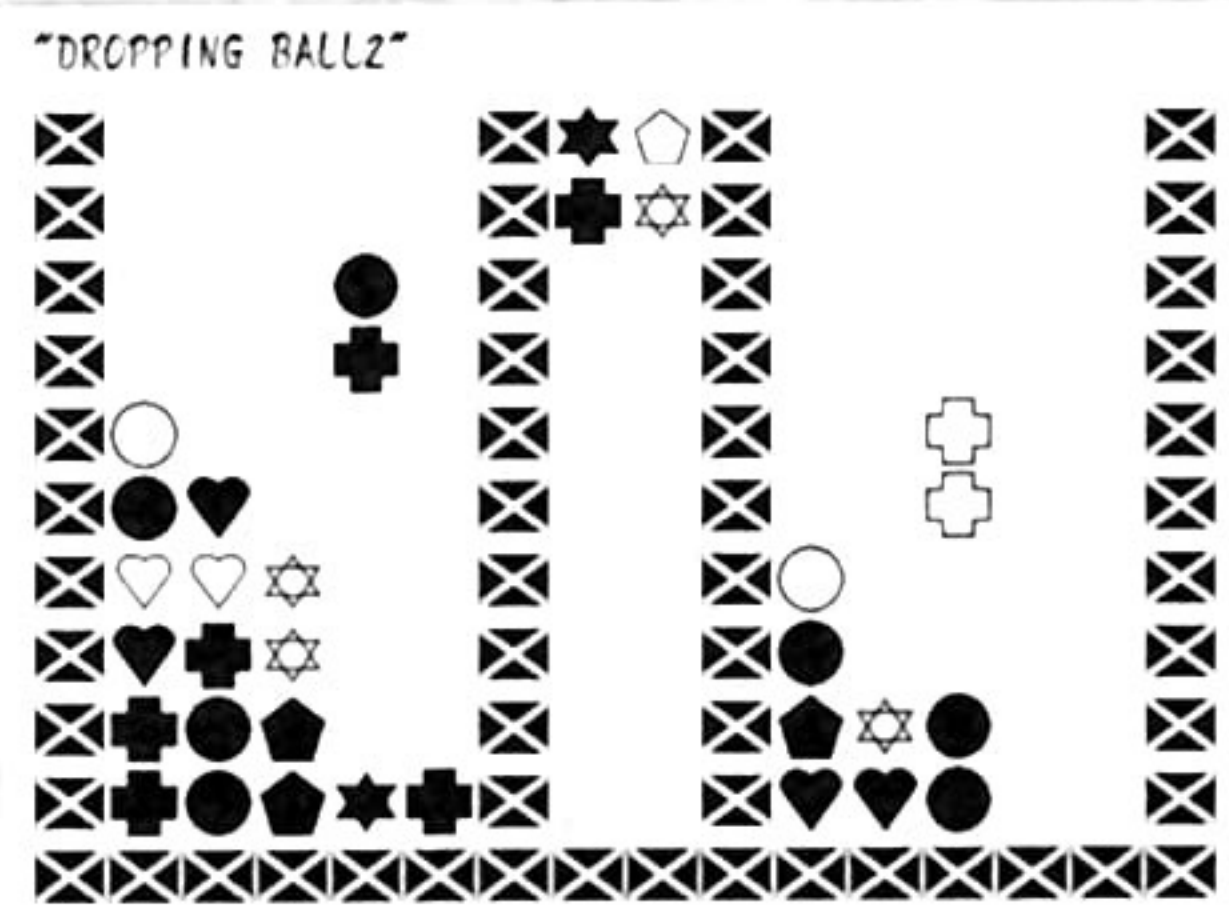
FOLLOWING CURSOR
→ p. 43
「遅れて追随するカーソル」
↑↓←→：カーソルの移動
ボタン0（Zキー）：ボールを入れ替える

Stage02　落とす　Drop

"DROPPING BLOCK"

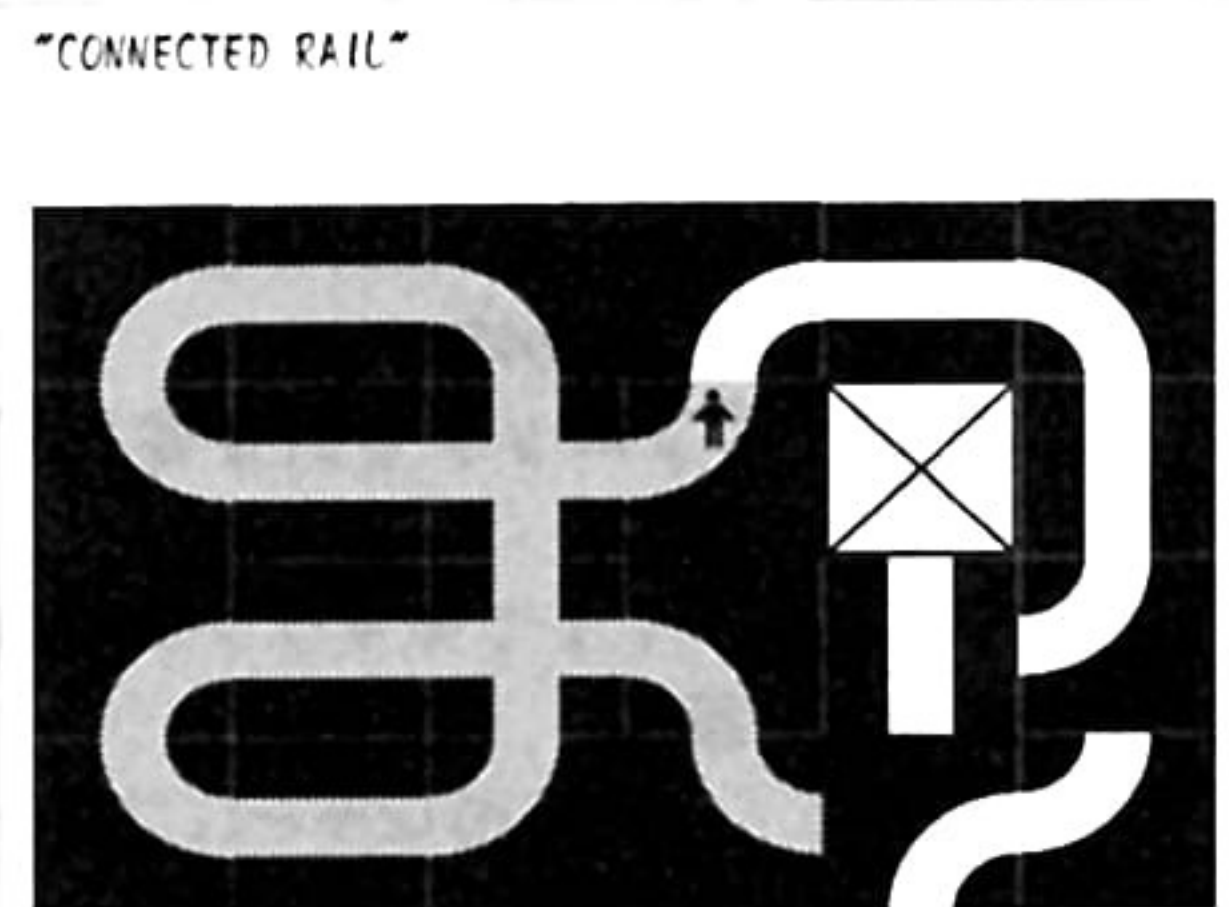
DROPPING BLOCK
→ p. 50
「ブロックを落とす」他
←→：ブロックの左右移動
↓：落下速度を上げる
ボタン0（Zキー）：ブロックの回転

"DROPPING JEWEL"

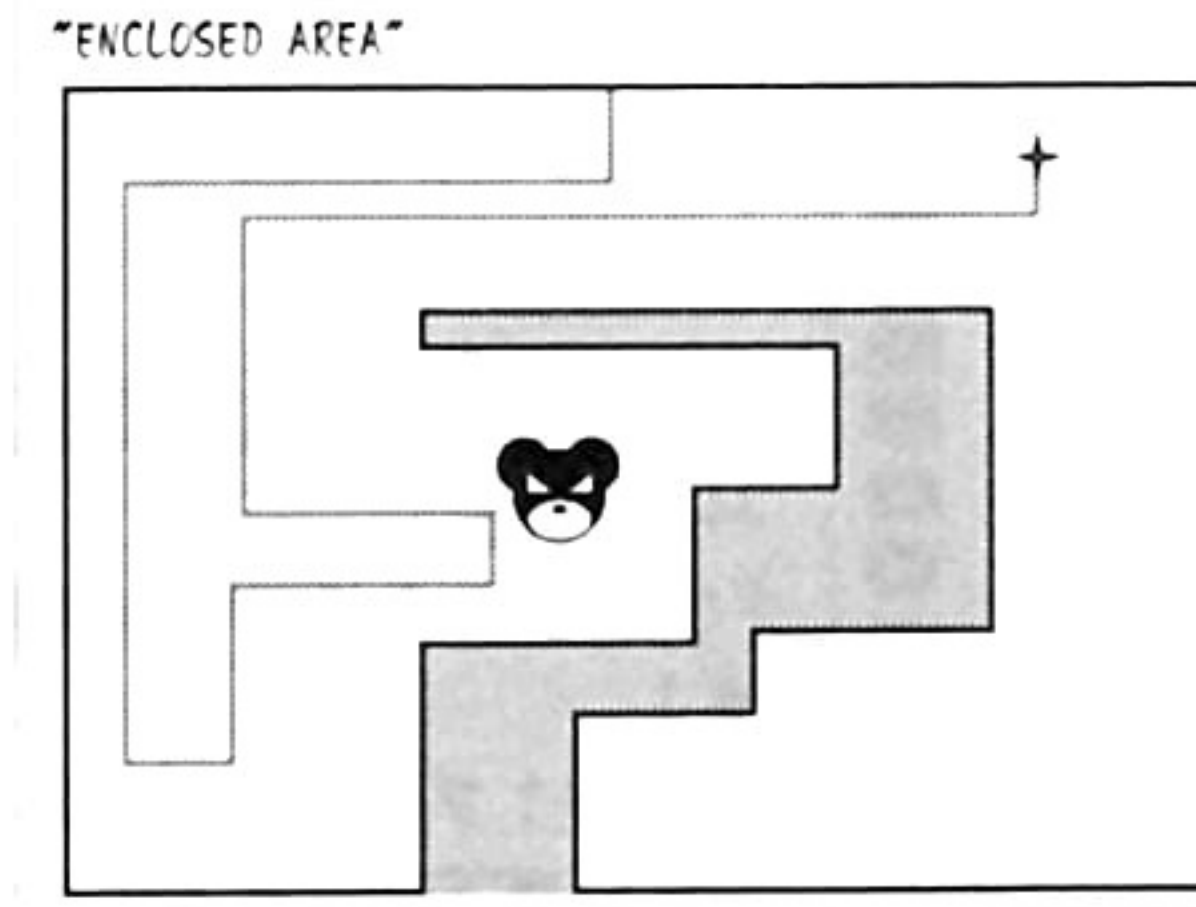
DROPPING JEWEL
→ p. 76
「宝石を落とす」他
←→：宝石の左右移動
↓：落下速度を上げる
ボタン0（Zキー）：宝石の並び順の変更

"DROPPING BALL"

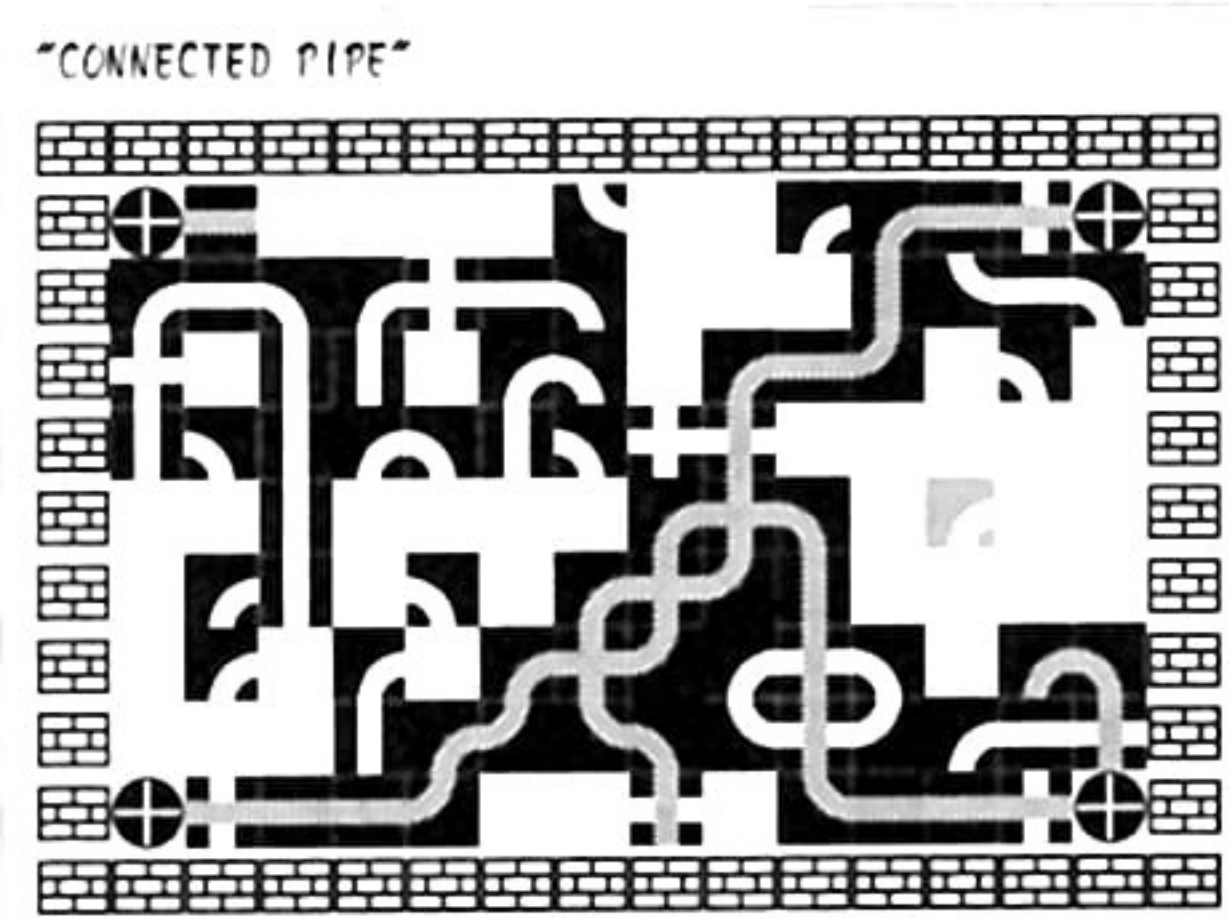
DROPPING BALL
→ p. 90
「ボールを落とす」他
←→：ボールの左右移動
↓：落下速度を上げる
ボタン0（Zキー）：ボールの回転

"DROPPING BLOCK 3D"

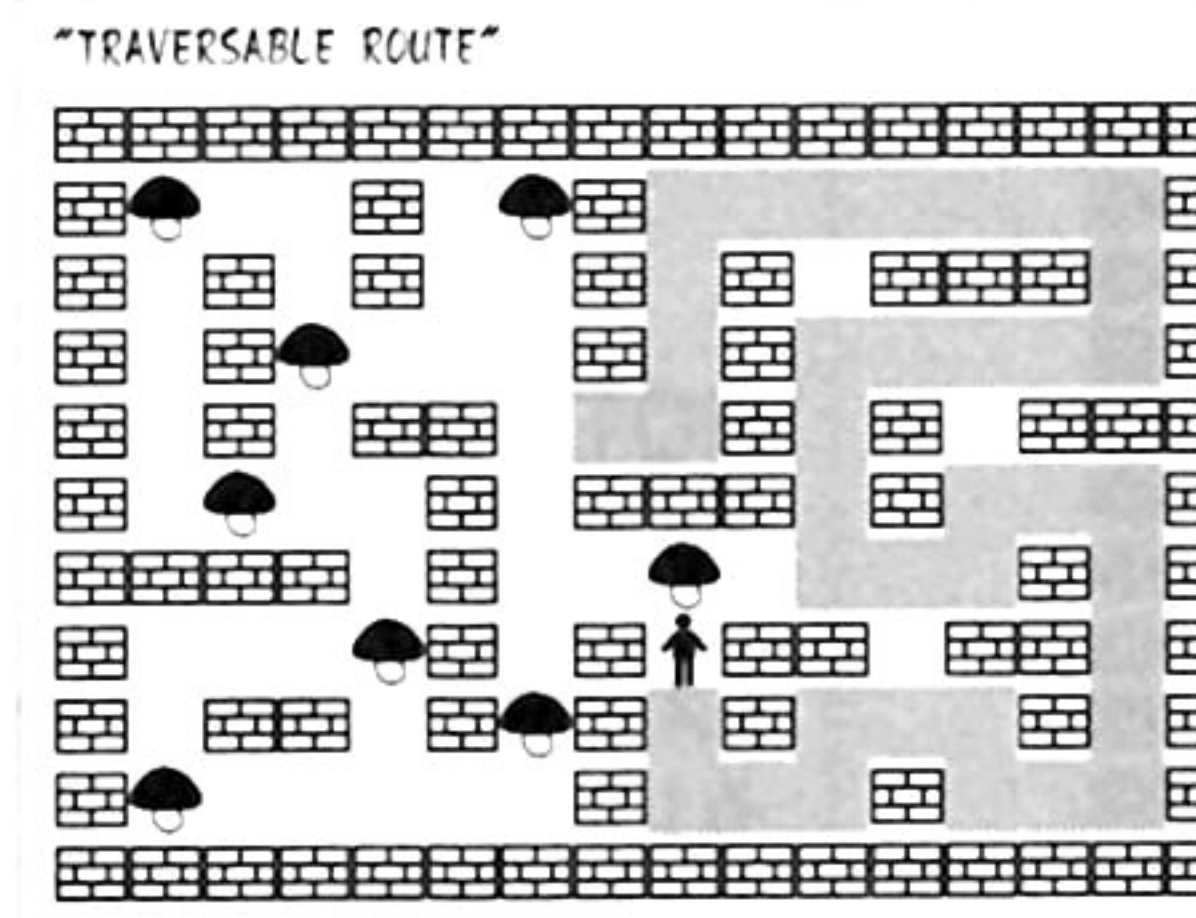
DROPPING BLOCK 3D
→ p. 123
「3次元のブロックを落とす」他
↑↓←→：ブロックの移動
ボタン3（Vキー）：落下速度を上げる
ボタン0〜2（Z、X、Cキー）：ブロックの回転

"DROPPING BALL2"

DROPPING BALL2
→ p. 119
「一度では消えないボール」
←→：ボールの左右移動
↓：落下速度を上げる
ボタン0（Zキー）：ボールの回転

Stage03 つなぐ　Connect

"CONNECTED RAIL"

CONNECTED RAIL
→ p. 140
「線路をつなぐ」他
↑↓←→：カーソルの移動

"ENCLOSED AREA"

ENCLOSED AREA
→ p. 174
「線で囲む」他
↑↓←→：カーソルの移動
ボタン0（Zキー）：線を引く

"CONNECTED PIPE"

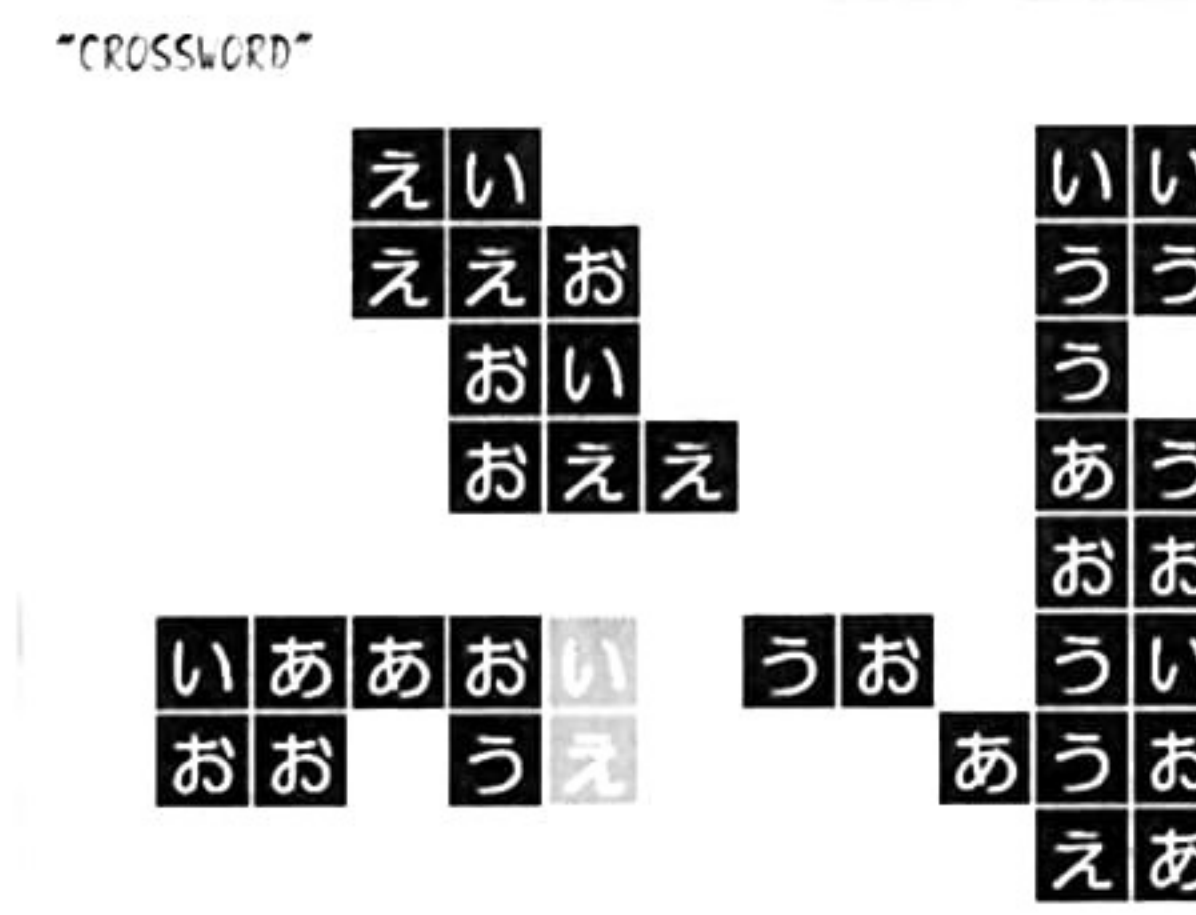
CONNECTED PIPE
→ p. 158
「パイプをつなぐ」
↑↓←→：カーソルの移動
ボタン0（Zキー）：パイプを置く

"TRAVERSABLE ROUTE"

TRAVERSABLE ROUTE
→ p. 187
「一筆書きでアイテムを回収する」
↑↓←→：カーソルの移動

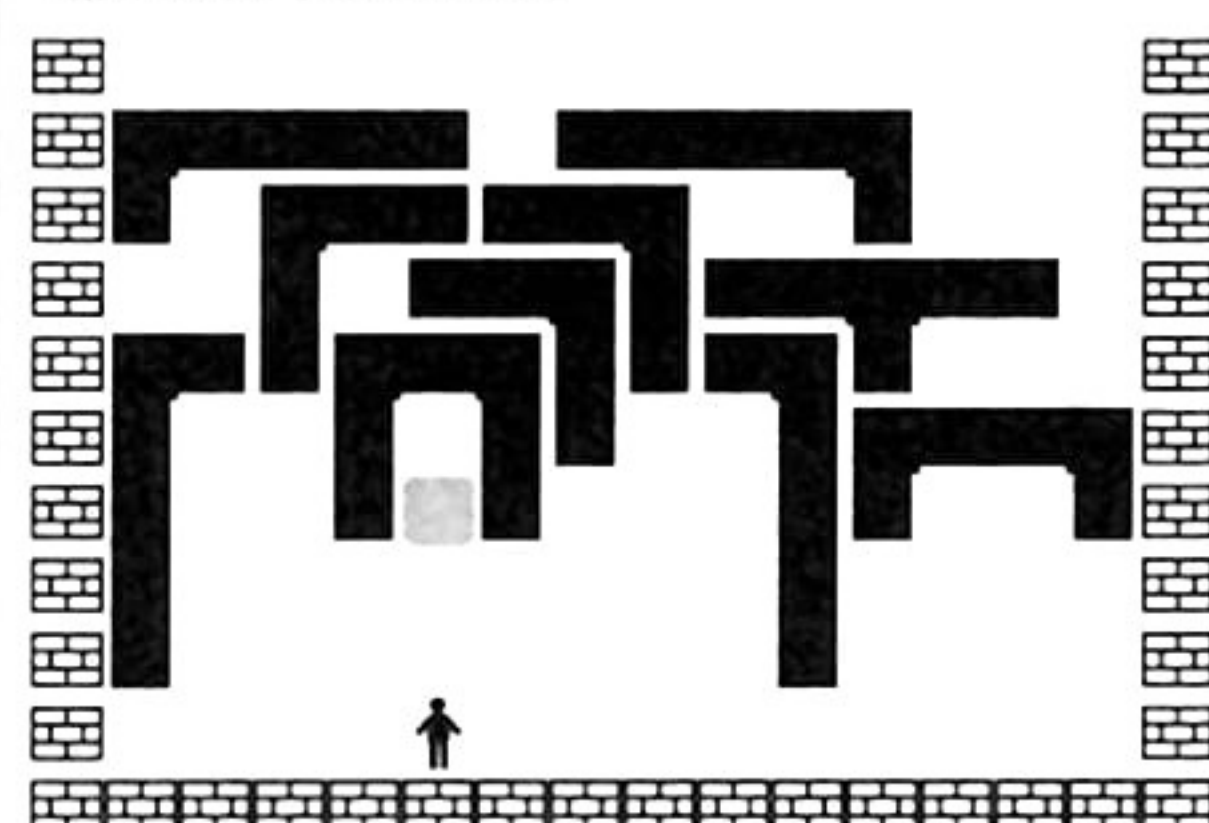
"LINKED SHAPE"

LINKED SHAPE
→ p. 166
「結合して形を作る」
↑↓←→：キャラクターの移動
ボタン0（Zキー）：分離する

"CROSSWORD"

えい
ええお
おい
おええ
いああお い
おお う え
うお
いい
うう
う
あう
おお
うい
あうお
えあ

CROSSWORD
→ p. 191
「言葉を作る」
↑↓←→：カーソルの移動
ボタン0（Zキー）：文字を置く

Stage04 ブロック Block

"RECTANGLE SHAPED BLOCK"

RECTANGLE SHAPED BLOCK

→ p. 200

「ブロックを矩形にして消す」

←→：キャラクターの移動

ボタン0（Zキー）：ブロックを撃つ

"SURROUNDING BLOCK"

SURROUNDING BLOCK

→ p. 234

「ブロックで囲んで消す」

←→：ブロックの移動

↓：落下速度を上げる

"TRANSFORMED BLOCK"

TRANSFORMED BLOCK

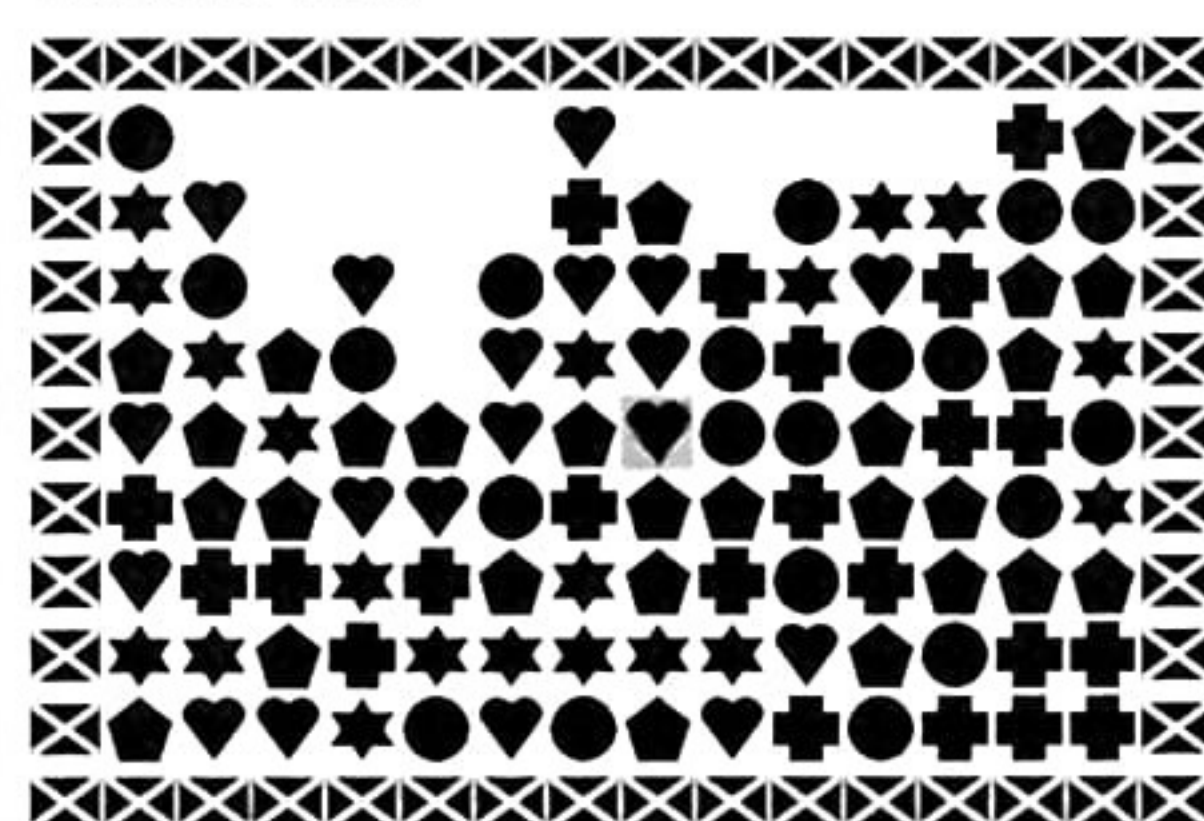
→ p. 209

「ブロックを変形させる」

←→：ブロックの移動

↑：上昇速度を上げる

ボタン0〜2（Z、X、Cキー）：ブロックを伸ばす

"CONNECTED BLOCK"

CONNECTED BLOCK

→ p. 241

「つながったブロックを消す」

↑↓←→：カーソルの移動

ボタン0（Zキー）：ブロックを消す

"STRUCK BLOCK"

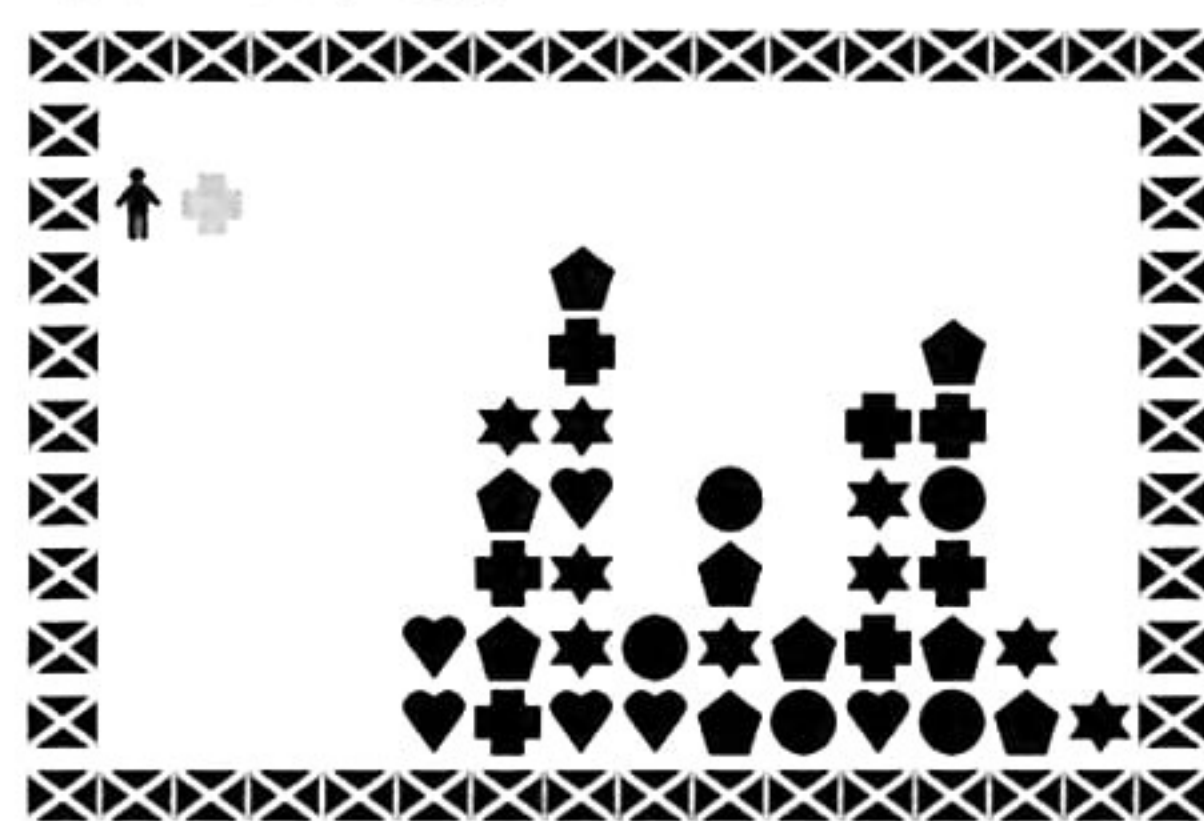
STRUCK BLOCK

→ p. 215

「ブロックをぶつけて壊す」

←→：ブロックの移動

↓：落下速度を上げる

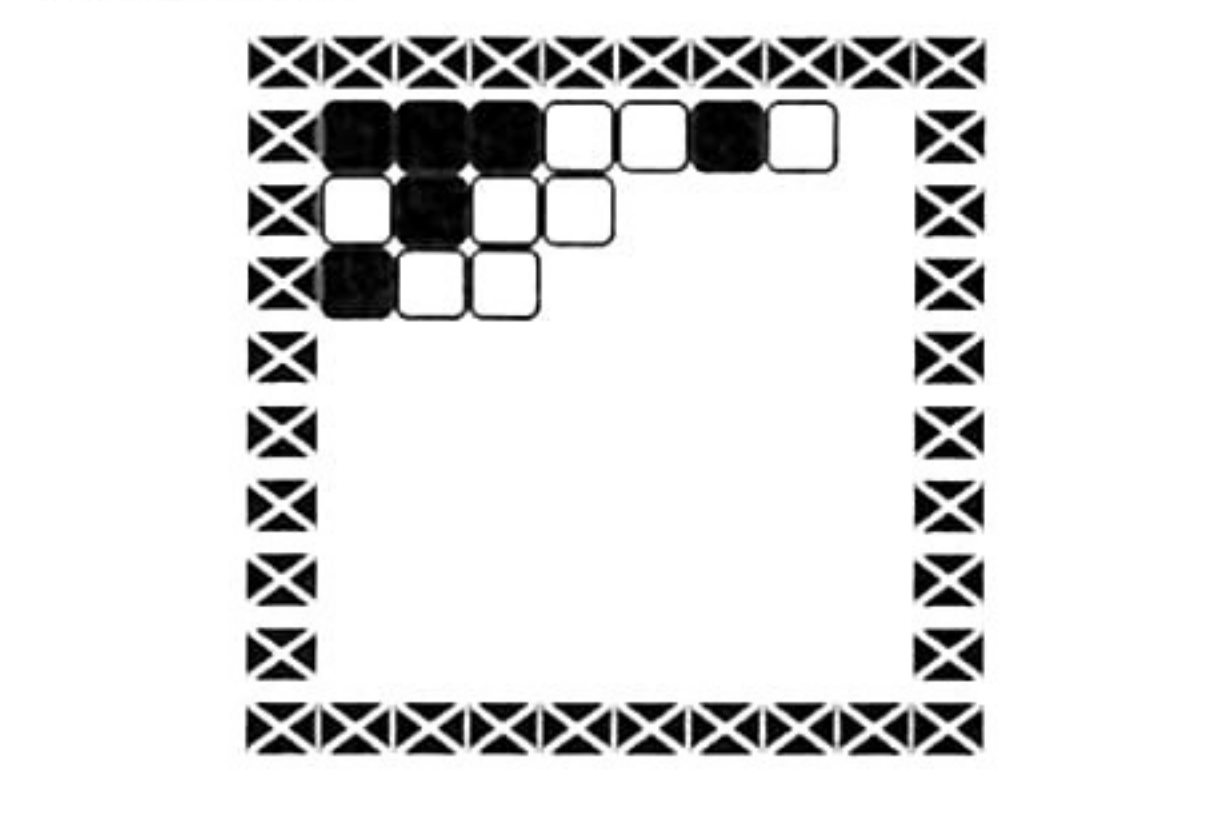
"DRAWN AND SHOT BLOCK"

DRAWN AND SHOT BLOCK

→ p. 248

「ブロックを引き寄せて撃つ」

↑↓：キャラクターの移動

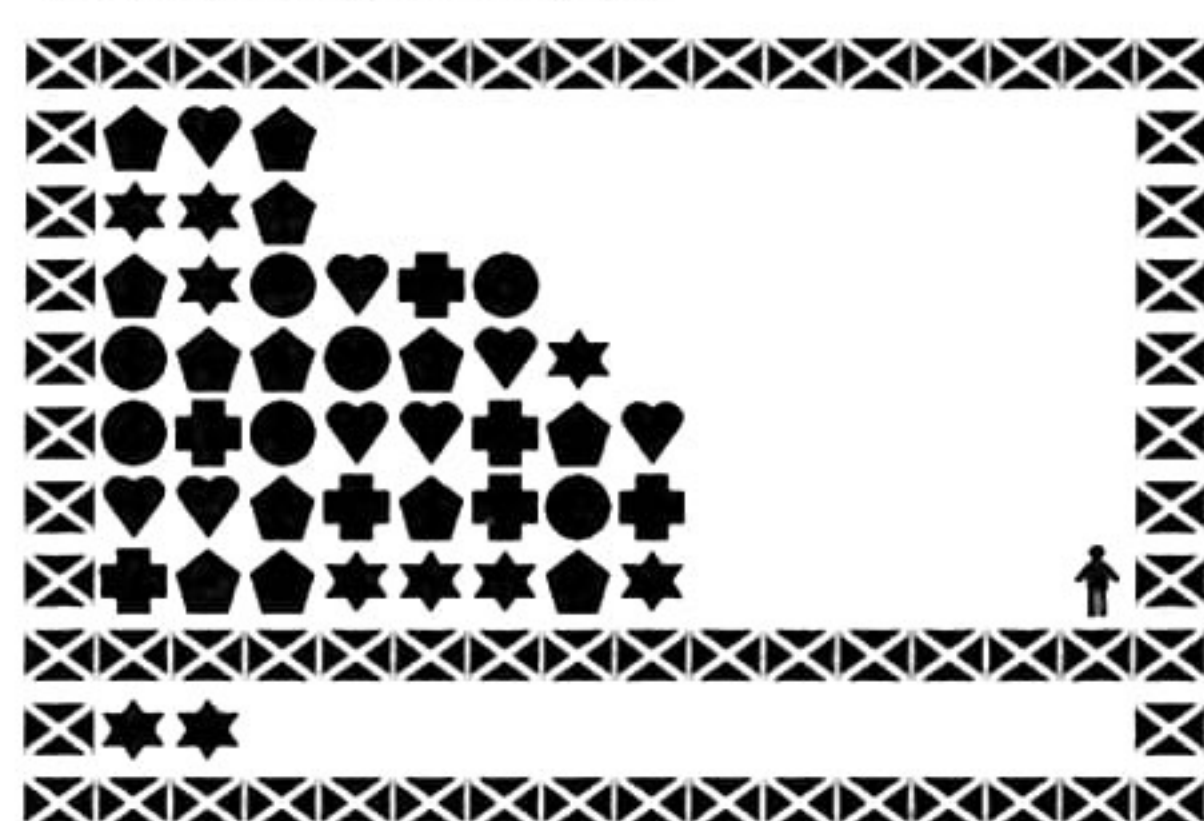
ボタン0（Zキー）：ブロックを引き寄せる（撃つ）

"ROTATED STAGE"

ROTATED STAGE

→ p. 223

「ステージを回転させる」

←→：ステージの回転

"PUSHED AND COLLECTED BLOCK"

PUSHED AND COLLECTED BLOCK

→ p. 257

「ブロックを突き落として集める」

↑↓：キャラクターの移動

ボタン0（Zキー）：ブロックを突き落とす

"FOOD BLOCK"

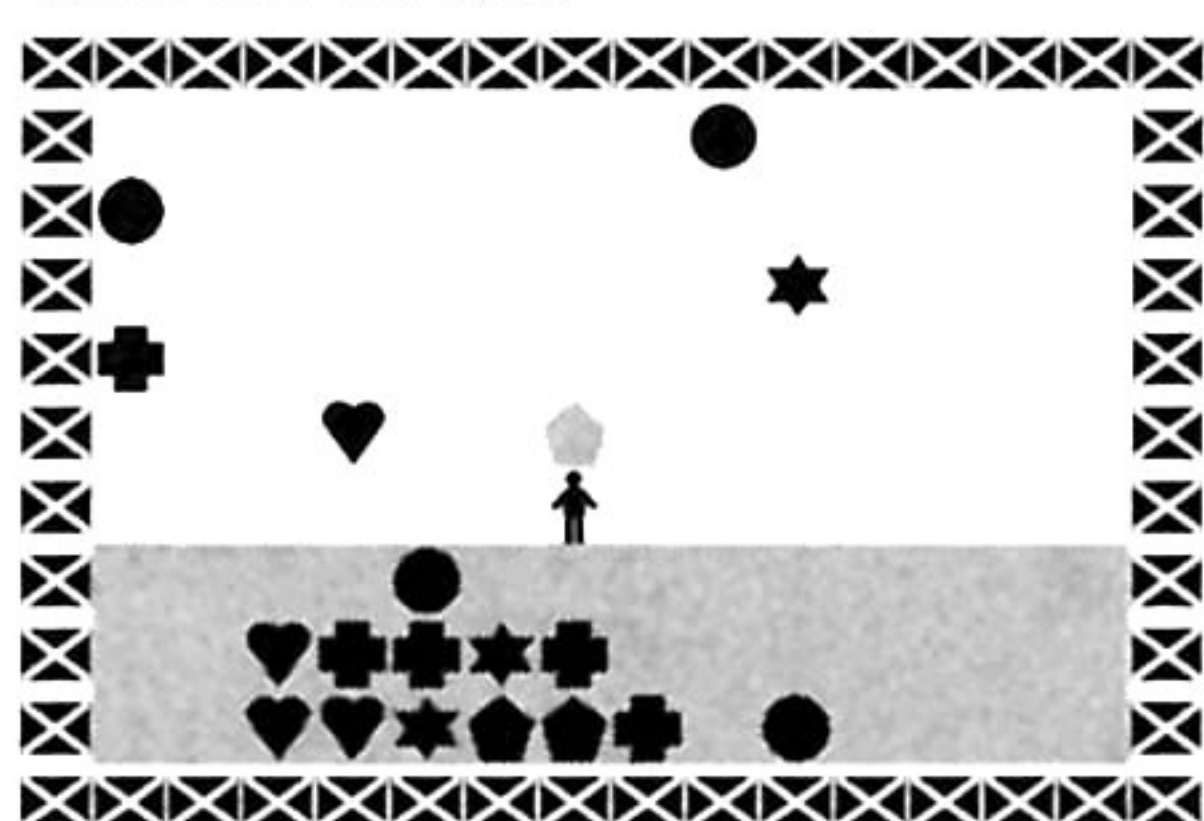
FOOD BLOCK

→ p. 227

「エサのブロックを消す」

←→：ブロックの移動

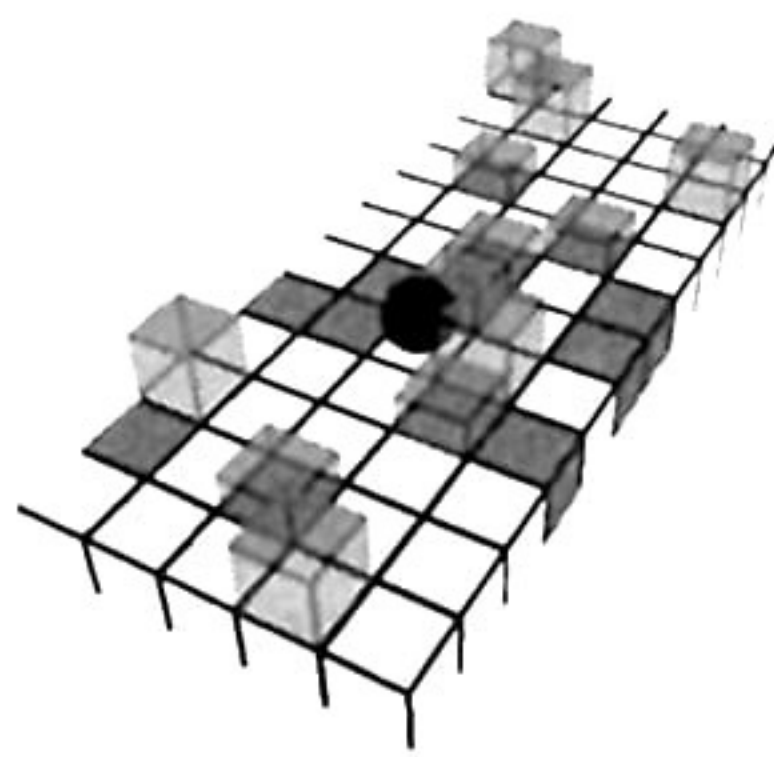
↓：落下速度を上げる

"CAUGHT AND PILED BLOCK"

CAUGHT AND PILED BLOCK

→ p. 262

「落ちてくるブロックを拾って積む」

←→：キャラクターの移動

ボタン0（Zキー）：ブロックを積む

MARKED AND SUNK BLOCK

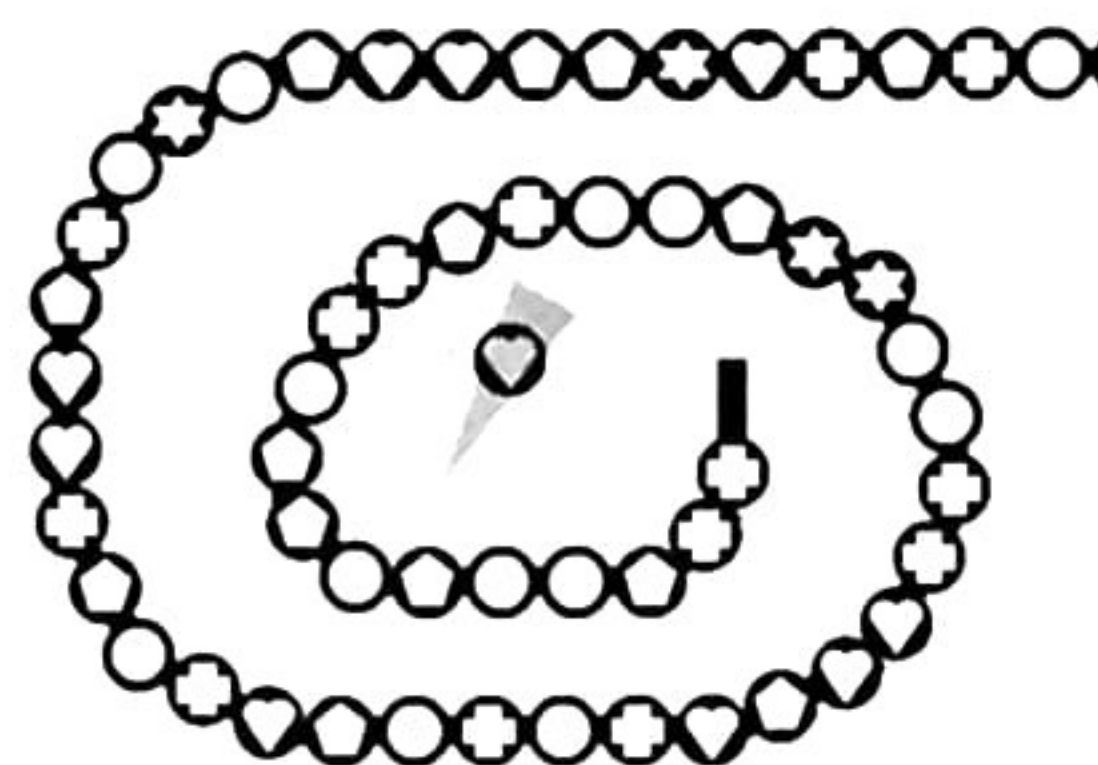
→ p. 270

「床をマークしてブロックを消す」

↑↓←→：カーソルの移動

ボタン0（Zキー）：床をマークする

ボタン1（Xキー）：ブロックを消す

Stage05　ボール　Ball

BALL ON RAIL

→ p. 278

「軌道に沿って進むボール」他

←→：砲台の回転

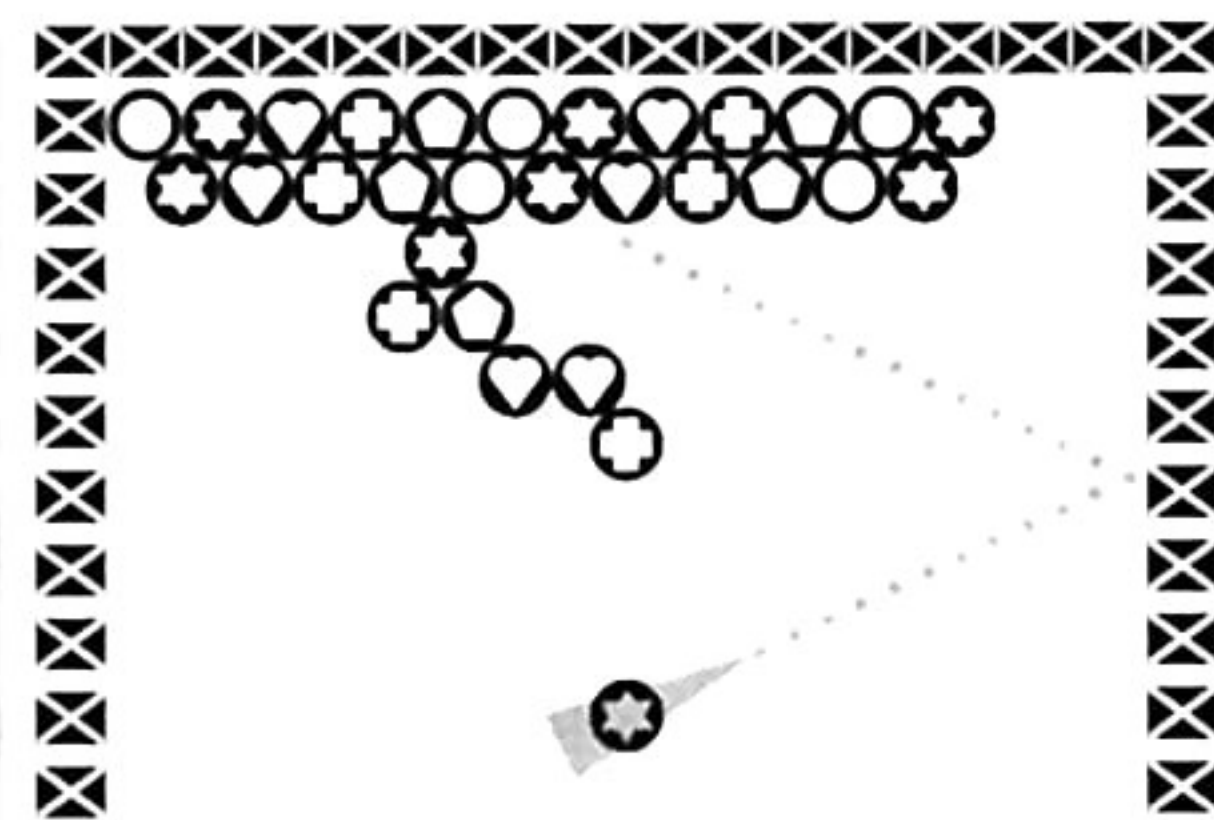
ボタン0（Zキー）：ボールを撃つ

SNAKE BALL

→ p. 337

「ボールをヘビ状に動かす」

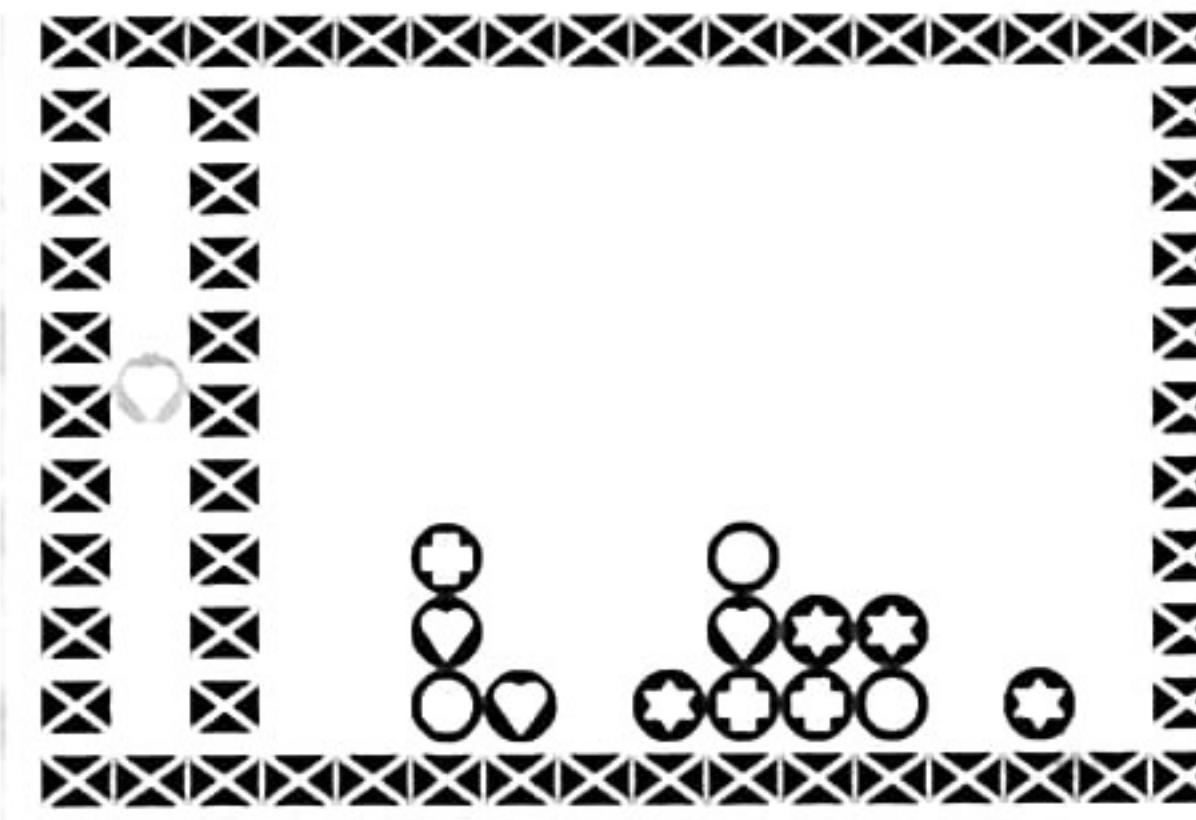
↑↓←→：ボールの移動

ボタン0（Zキー）：ボールの固定

HANGING BALL

→ p. 301

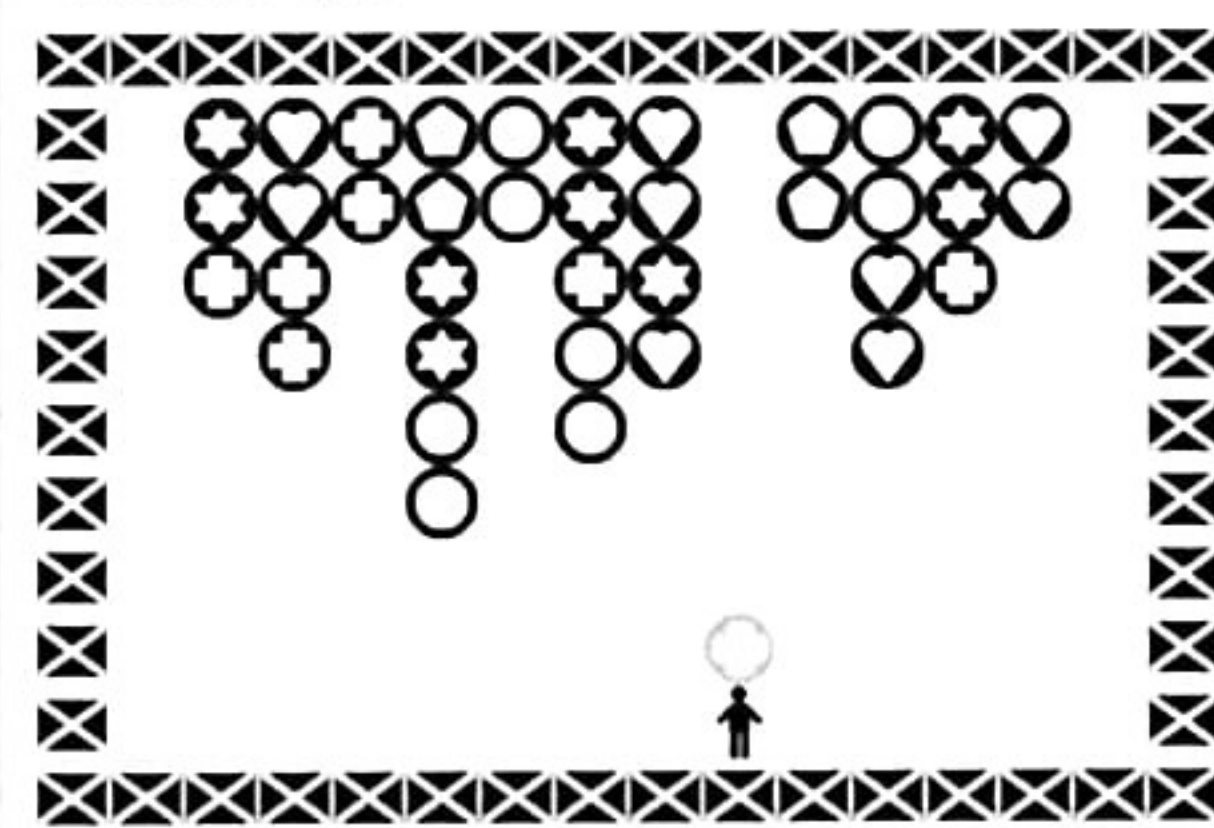
「ぶら下がったボール」他

←→：砲台の回転

ボタン0（Zキー）：ボールを撃つ

FLIPPED BALL

→ p. 345

「ばねでボールを撃つ」

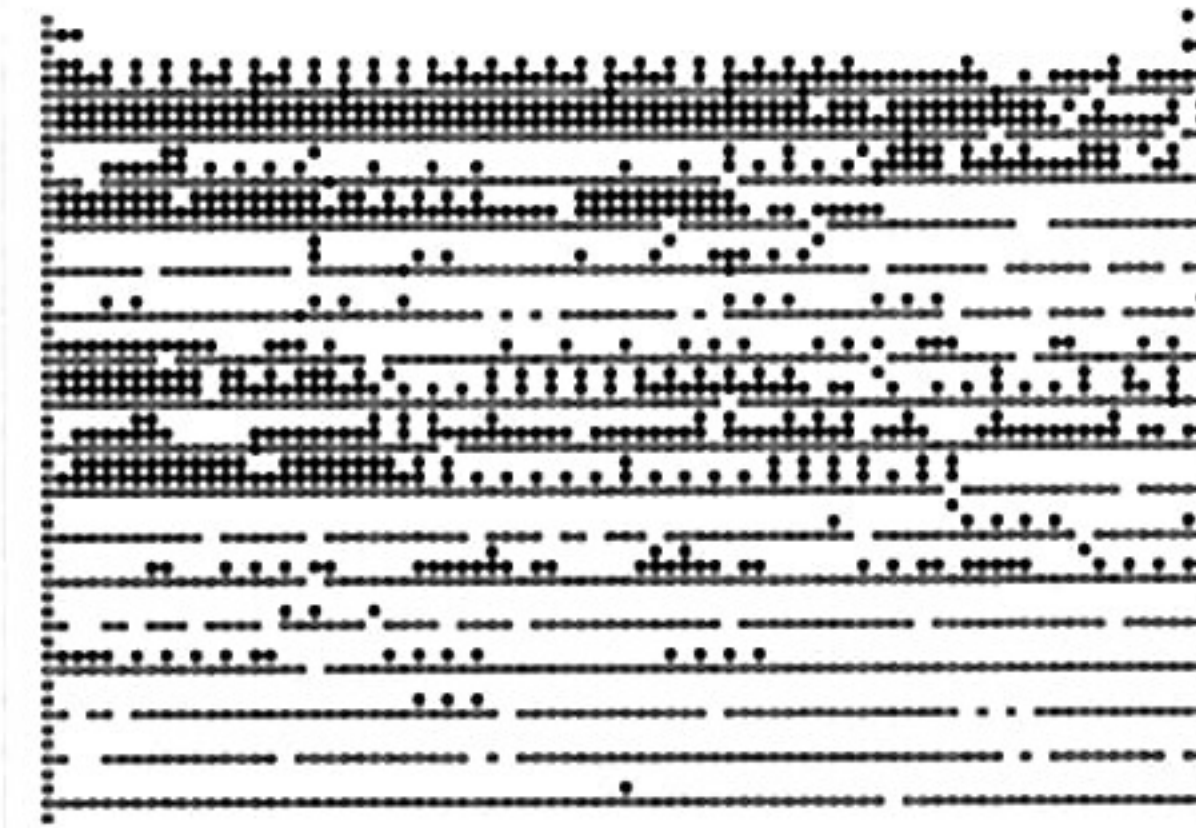
↓：ボールを下げる

COLLECTED BALL

→ p. 322

「ボールを拾って集める」

←→：キャラクターの移動

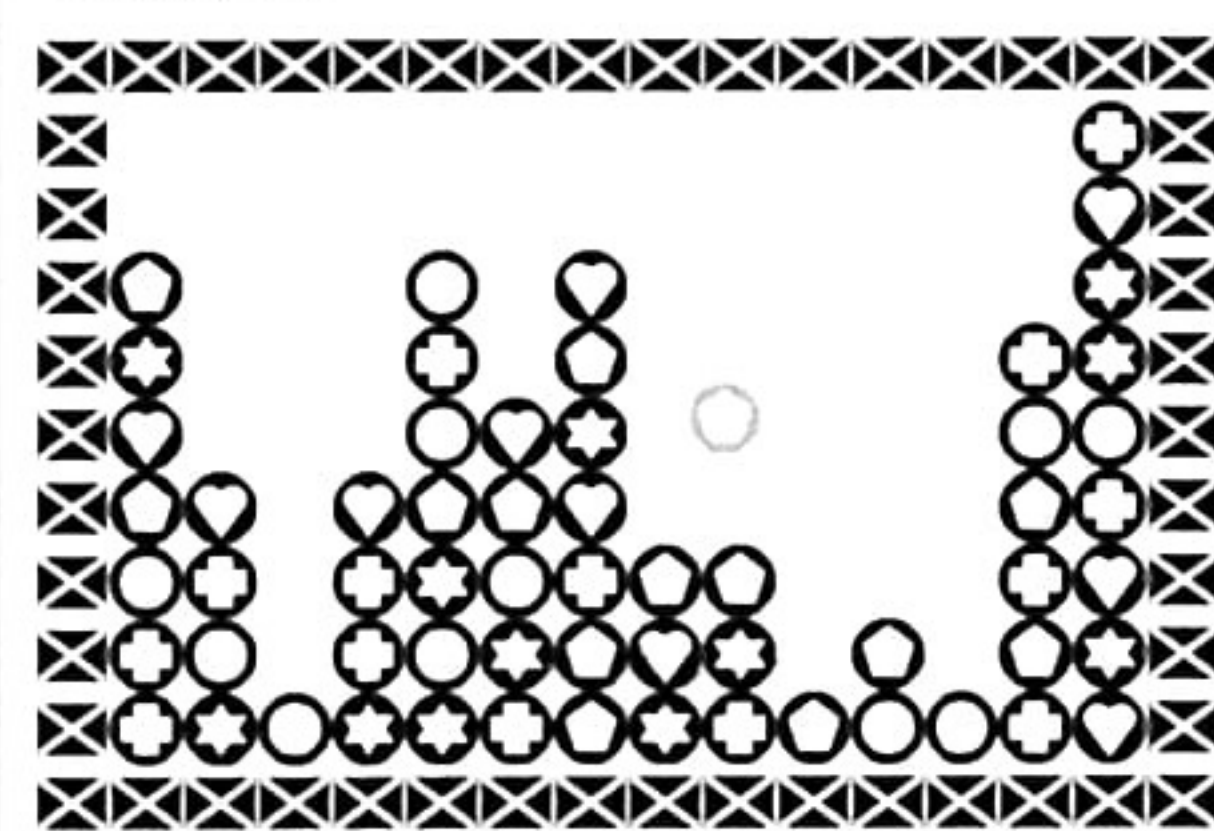
ボタン0（Zキー）：ボールを拾う

ボタン1（Xキー）：ボールを戻す

ROLLING BALL

→ p. 352

「転がる大量のボール」

ボタン0（Zキー）：左上のボールを出す

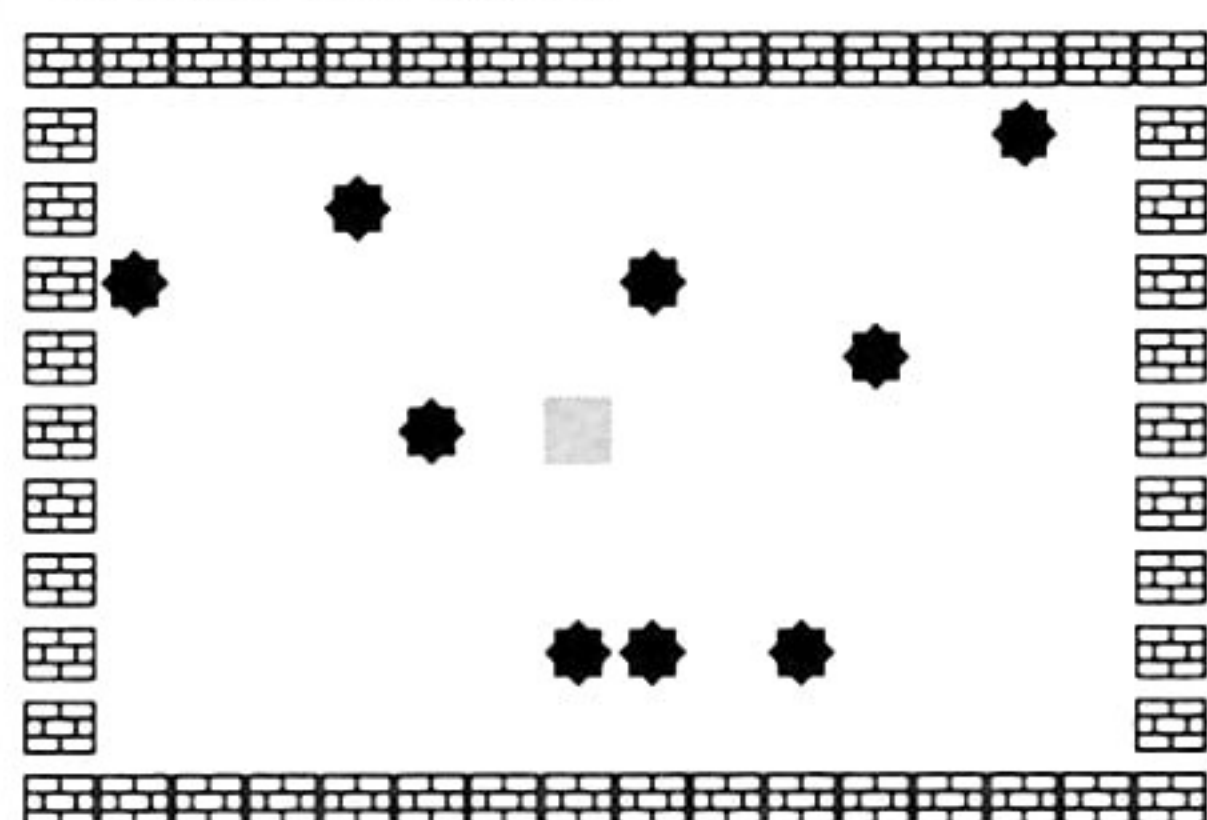
ボタン1（Xキー）：右上のボールを出す

SWAPPED BALL

→ p. 331

「ボールを入れ替える」

↑↓←→：カーソルの移動

ボタン0（Zキー）：ボールを拾う（入れ替え）

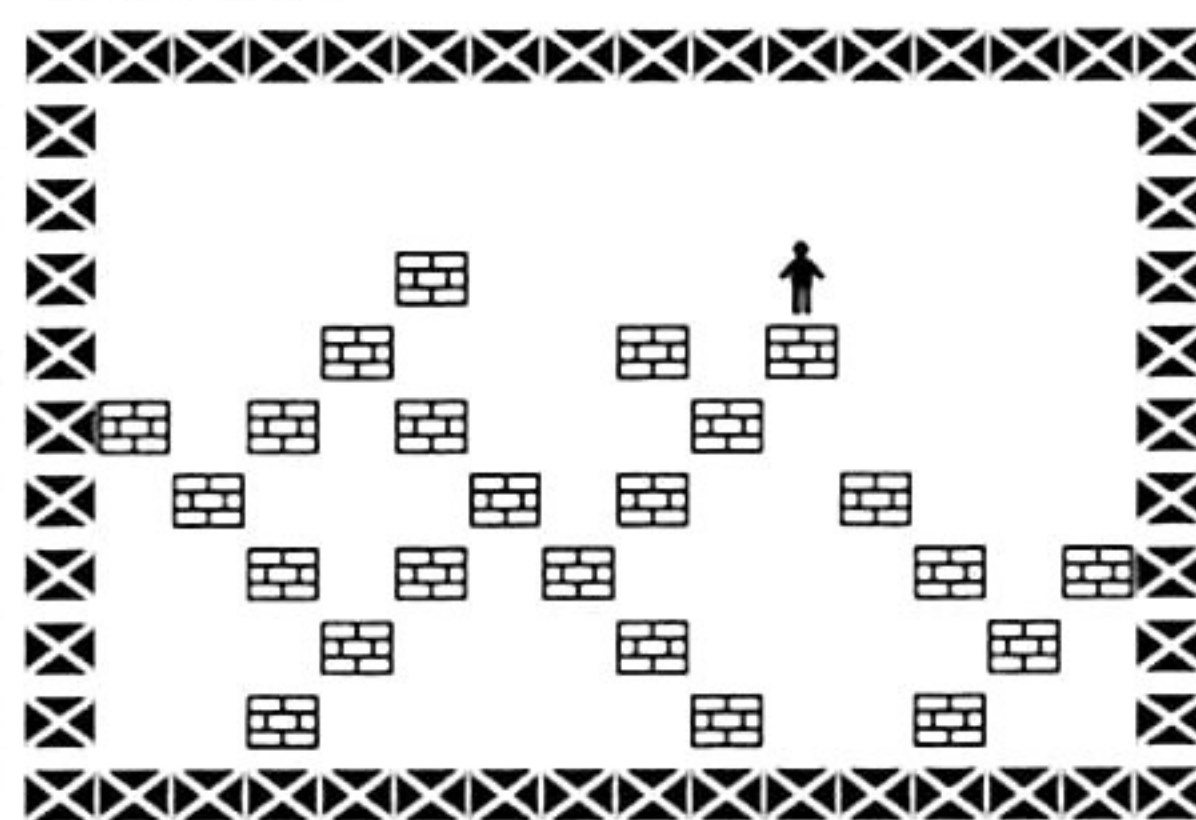
Stage06 その他 Others

MEMORIZING ITEM POSITION

→ p. 358

「アイテムの位置を記憶する」

↑↓←→：カーソルの移動

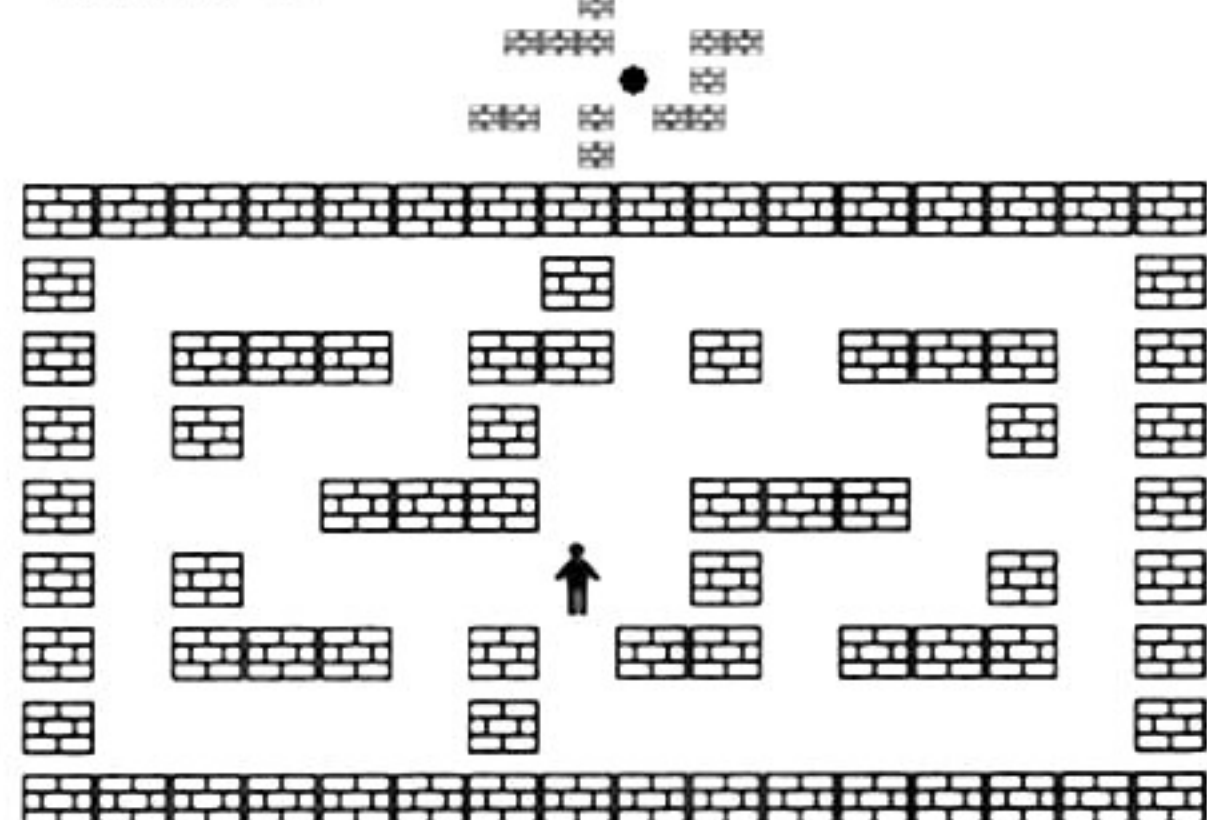
ボタン0（Zキー）：アイテムの回収

ボタン1（Xキー）：アイテムの表示

MAKING FLOOR

→ p. 369

「床を作って進む」

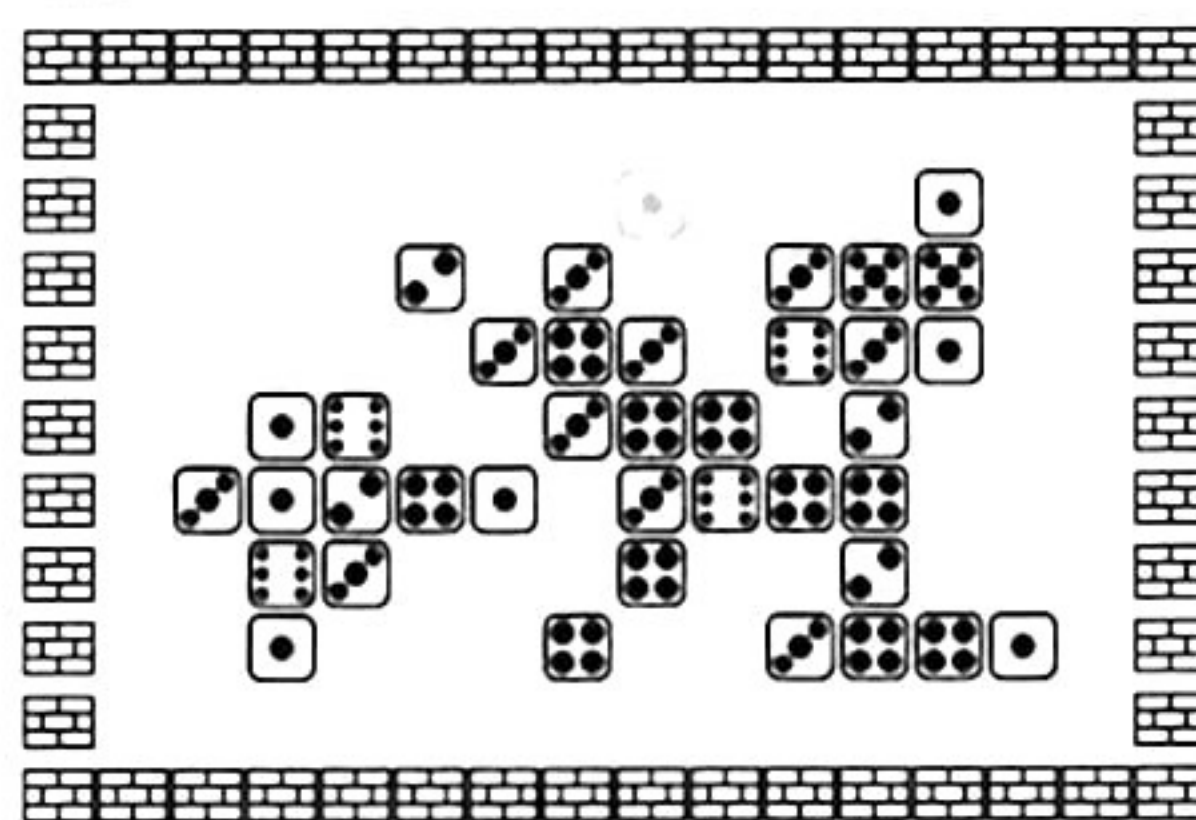
←→：キャラクターの移動

ボタン0（Zキー）：床を作る（壊す）

TREASURE MAP

→ p. 361

「地図を頼りにアイテムを探す」

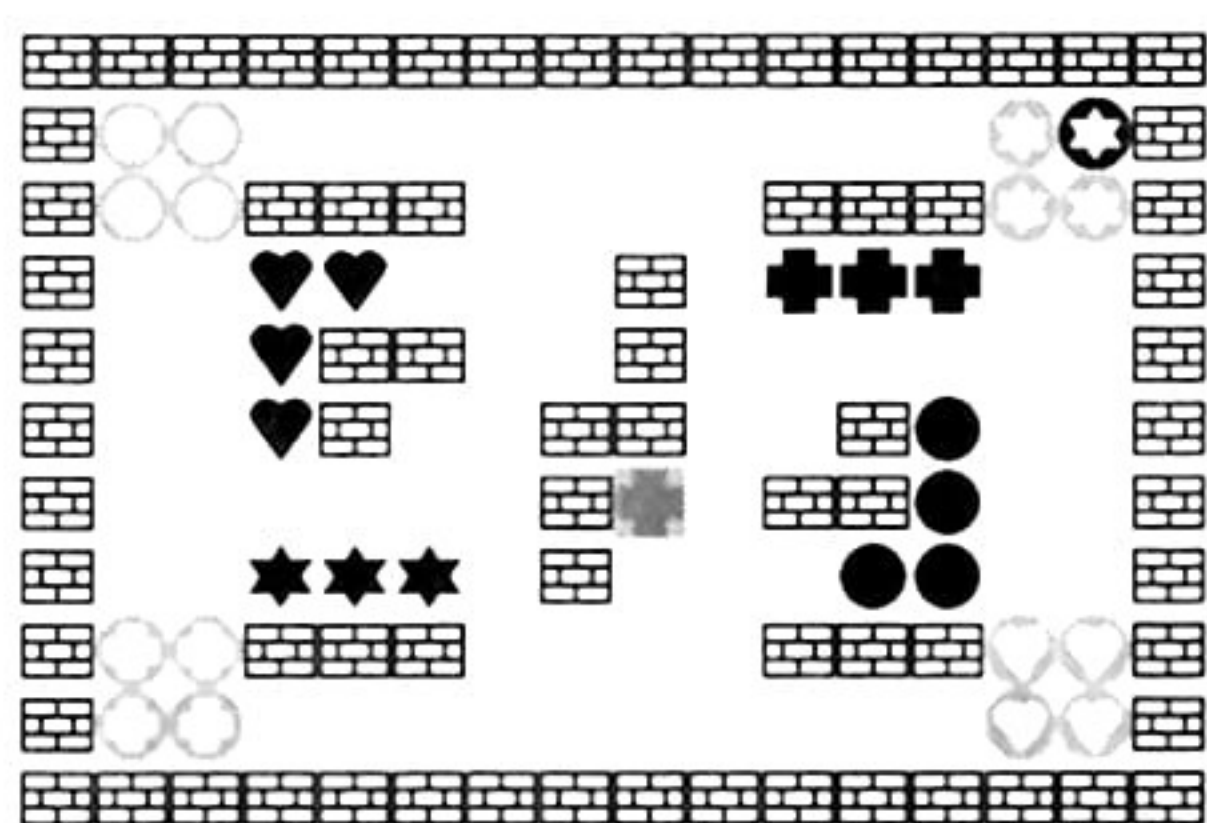
↑↓←→：キャラクターの移動

ボタン0（Zキー）：アイテムの回収

DICE

→ p. 373

「サイコロを揃えて消す」

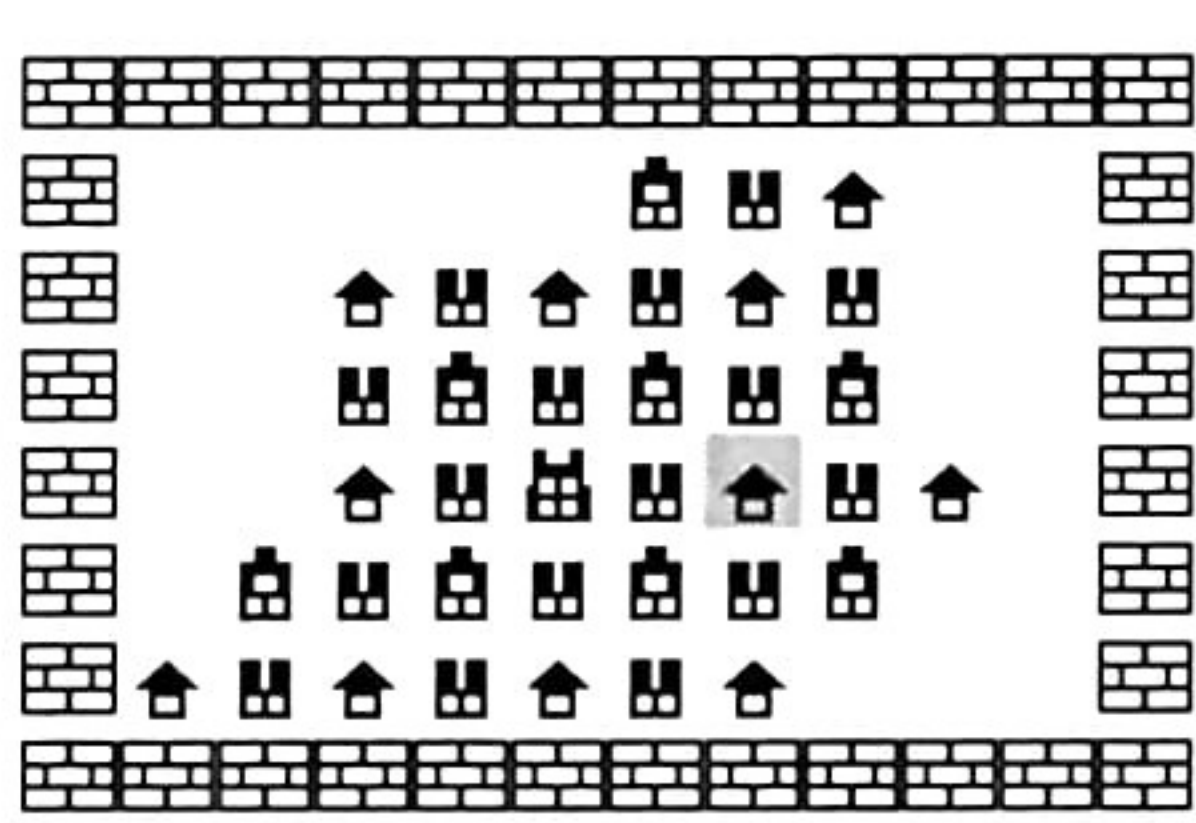
↑↓←→：カーソルを動かす

ボタン0（Zキー）：サイコロを置く

DELIVERING LOAD

→ p. 365

「荷物を指定の場所に運ぶ」

↑↓←→：カーソルの移動

ボタン0（Zキー）：荷物を拾う

BUILDING

→ p. 379

「建物を建てる」

↑↓←→：カーソルを動かす

ボタン0（Zキー）：建物を建てる（消す）

引用ゲーム一覧　Appendix 02

本文中で引用したゲームの一覧表です。PCや家庭用ゲーム機に移植されている作品も多いので、気になるゲームはぜひ一度、実際に遊んでみることをお勧めします。なお、一部のゲームには続編が出ていたり、移植の際にタイトルが変わっていたりする場合があるので、詳しくは各メーカーのWebサイトやショップなどでお確かめください。

※凡例

■ゲーム名

・メーカー：発売年度：プラットフォーム

AC：アーケード、DC：ドリームキャスト、DS：ニンテンドーDS
GBA：ゲームボーイアドバンス、GC：ゲームキューブ
PC：PC/AT互換機（Windowsパソコン）、PS：プレイステーション
PS2：プレイステーション2、PS3：プレイステーション3
PSP：プレイステーションポータブル、SS：セガサターン
WII：Wii（バーチャルコンソールを含む）、XB：Xbox、XB360：Xbox360

・本書のなかで取り上げた主な特色

・コメント

■I.Q（あいきゅー）

・ソニー・コンピュータエンタテインメント：1997：PS, PS2, PSP

・床をマークしてブロックを消す

ステージの奥から転がってくるキューブ（立方体）を消すゲームです。床をマークしておき、キューブがマークの上に乗ったときに、タイミングよくボタンを押すと、キューブを沈めて消すことができます。キャラクターがキューブにつぶされる他、沈めてはいけないキューブも登場するので、反射神経と思考力の両方が求められます。

■アクアラッシュ

・ナムコ：1999：AC

・ブロックを変形させる

ステージ上方から降ってくるブロックを消すゲームです。下から新しいブロックをはめ込んで、ブロックを横1段に揃えると、その段を消すことができます。『テトリス』に似ていますが、ボタン操作で新しいブロックを変形させられることが特徴です。

■ウェルトリス

・ビデオシステム：1991：AC

・ブロックを落とす

『テトリス』の原作者による、テトリスの2.5次元版といった雰囲気のゲームです。上下左右の4方向に壁があり、ブロックは壁に沿って、手前から奥に向かって落ちていきます。奥の平面上に、ブロックを縦1列または横1列

に揃えると消すことができます。上から下に積み上げるだけのテトリスとは違い、4方向からブロックを詰めることができるので、戦略が広がる一方、難易度も上がっています。

■魚ポコ（うおポコ）

・ケイブ：1998：AC

・ばねでボールを撃つ

落ち物パズルゲームの一種ですが、ばねでボールを撃つことが特徴です。レバーを下に入れるとばねを引っ張り、レバーを中央に戻すとボールを撃ちます。ばねを引く強さによって、ボールの落下位置が変わります。『戦球』と同じように、ボールは互い違いに積み上がるため、着地したボールが左右に転がりながら落ちていきます。シンプルな操作方法と、ボールの動きが楽しいゲームです。

■ヴォルフィード

・タイトー：1989：AC, PS, PS2

・線で囲む

『クイックス』の続編です。自機を動かして線を引き、領域を囲んで塗りつぶします。一定割合以上の領域を塗りつぶすと、ステージクリアとなります。内容はクイックスと同じですが、グラフィックが向上しており、敵の種類も増えています。また、アイテムが出現します。アイテムを囲むと回収することができます。アイテムには自機のスピードが上がるものや、レーザーを撃てるようになるものなどがあります。

■エメラルディア

・ナムコ：1993：AC

・ブロックをぶつけて壊す

落ちてくるブロックを、積み上げたブロックにぶつけて壊すゲームです。1回ぶつけると、ブロックにひびが入ります。ひびが入ったブロックに、もう1回ブロックをぶつけると、ブロックを壊して消すことができます。ブロックをぶつけた衝撃が、隣接するブロックにも伝わることを利用すると、広範囲のブロックをまとめて壊すことができます。

■オーマイガァー！

・アトラス：1993：AC

・ボールをヘビ状に動かす

ボールを使った落ち物パズルゲームの一種です。ボールを積み上げ、同じ種類のボールを並べて消します。本作品の特徴は、ステージ上方に出現するボールが、ヘビ状に連なっていることです。先頭のボールを動かすと、残りのボールが先頭のボールを追いかけるように動きます。一度に4個や5個といった多数のボールが現れることがあり、それがヘビ状に動くので、ボールを思いどおりの場所に落とすのには慣れが必要です。

■オトスタツ

・ソニー・コンピュータエンタテインメント：2002：PS2

・建物を建てる

大きな建物を数多く建てることが目的のゲームです。ステージに「土」「水」「木」を特定の配置で置くと、建物が建ちます。さらに建物を特定の配置にすると、建物で囲まれた土地に、より大きな新しい建物が建ちます。建物が建つ法則は、ライフゲームに似ています。建物は折り紙細工のような立体的なグラフィックで表現されており、高いタワーが建つと花火やサーチライトの演出があるなど、見た目が派手で、内容もテンポがよく楽しいゲームです。

■カチャット

・タイトー：1993：AC

・パイプをつなぐ

ステージ上方から次々に降ってくるパイプを、つなげて消していくゲームです。積み上がったパイプは回転させることができます。パイプを回転させてルートを作り、ステージの左右に並んでいる水源同士をつなぐと、パイプが消えます。パイプが降ってくるスピードが速く、ステージがあっという間に埋まってしまうので、スリルのあるゲームです。

■ガッタンゴットン

・コナミ：1982：AC

・線路をつなぐ、線路に沿って進むキャラクター、キャラクターの進路を予測して表示する

機関車が線路から落ちないように、乗客のところまで導くゲームです。格子状に敷き詰められた線路を、15パズルのようにスライドさせて、配置を変えることができます。機関車の予測進路が表示されるので、これを参考に上手にルートを作り、敵に当たらないように乗客を回収します。線路には直線やカーブの他、十字路や、ランダムな方向に進む分岐もあります。

■ギャルズパニック

・カネコ：1990：AC, PC

・線で囲む

『クイックス』に似たゲームです。カーソルを動かして線を引き、領域を囲んで塗りつぶします。ステージ内の一定割合以上を塗りつぶすとクリアとなることや、さまざまな敵が出現することなどは、クイックスと同じです。本作品の最大の特徴は、背景に女の子のグラフィック（大人向け）が表示され、ステージを塗りつぶすごとにグラフィックが見えるようになっていく点です。必死で敵を避けながら、女の子のグラフィックを開いていく様子には、独特の雰囲気があります。

■キューブリック

・コナミ：1989：AC

・線路をつなぐ

『ガッタンゴットン』に似たゲームです。キャラクターが道から落ちないように、パネルを15パズルのようにスライドさせて、ルートを作ります。キャラクターが通った道は消えます。ステージ内のすべての道を消すことが目的です。ステージごとに道がいろいろな形に配置されていて、次々に問題を解いていく感覚で、ステージをクリアしていきます。

■クイックス

・タイトー：1981：AC, PS, PS2, PSP

・線で囲む、囲んだ領域を塗りつぶす、囲まれた領域を避けて動く敵

カーソルを動かして線を引き、領域を囲んで塗りつぶすゲームです。ステージ内の一定割合以上の領域を塗りつぶすと、ステージクリアとなります。ステージには、塗りつぶされていない領域を動き回るものや、線に沿って追いかけてくるものなど、いろいろな敵が出現します。敵に接触しないように、ステージを効率よく塗りつぶしていくことが、ゲームの目的です。

■クォース

・コナミ：1989：AC, PS2

・ブロックを矩形にして消す

降ってくるブロックに新しいブロックを撃ち込み、矩形にすることによって、ブロックを消すゲームです。降ってくるブロックは、カギ型、コの字型、T字型など、いろいろな形をしています。独立しているブロックでも、矩形状にすることによって、まとめて消すことができます。

■ぐっすんおよよ

・アイレム：1993：AC, PS, SS, WII

・自律的に動くキャラクター、ブロックを落とす

『レミングス』に落ち物パズルゲームを組み合わせたようなゲームです。ステージ上方から落ちてくるブロックを、左右に動かしたり回転させたりしながら、積み上げます。積み上げたブロックでルートを作り、自律的に動くキャラクターをゴールまで導くことがゲームの目的です。ブロックで敵をつぶしたり、爆弾で不要なブロックを壊したり、落下中のブロックにキャラクターを乗せたりと、幅広いアクションが楽しめます。

■クラックス

・アタリ：1989：AC, PS2

・落ちてくるブロックを拾って積む

転がってくるタイルを受け止めて、別の領域に積み上げ、同じ種類のタイルを並べて消すゲームです。タイルを積み上げて消す部分は『コラムス』に近いのですが、次々に転がってくるタイルを受け止めるアクションが加わることで、違ったゲーム性が生まれています。タイルが疑似3Dで表現されていて、画面奥から転がってくる点も独特です。

■ぐるりん

・フェイス：1994：AC

・ステージを回転させる

落ちてくるキャラクター（ブロック）を積み上げ、同じ種類のキャラクターを揃えて消すゲームです。ここまではオーソドックスな落ち物パズルゲームですが、この作品の特徴はステージを左右に回転できることです。回転によって積み上げたキャラクター全体が動き、意外な場所で同じ種類が揃って消えます。

■クレオパトラフォーチュン

・タイトー：1996：AC, DC, PS, PS2, SS

・ブロックで囲んで消す

降ってくる財宝とブロックを消すゲームです。財宝とブロックには、それぞれ小さなものと大きなものがあり、混じって降ってきます。ブロックで財宝を囲むと、消すことができます。ブロックを消すには、ブロックを隙間なく横1段に並べます。ステージ内の財宝とブロックをすべて消すと、キャラクター「パトラ子」のデモが楽しめますが、全消しを狙うのはなかなか難しいゲームです。

■コズモギャング・ザ・パズル

・ナムコ：1992：AC, GBA, PS, WII

・エサのブロックを消す

落ちてくるブロックや敵を次々と消していく、落ち物パズルゲームです。ステージ上方からは、ブロック、ボ

ール、そして「コズモ」と呼ばれる敵が降ってきます。ブロックは横一段に揃えると消えます。ボールは着地すると、ボールの表面に記された矢印の方向に進み、進路に積まれているコズモを弾き飛ばして消します。

■ことばのパズル もじぴったん

・ナムコ：2001：AC, DS, GBA, PS2, PSP, WII

・言葉を作る

ステージに文字を並べて、クロスワードのように縦横に言葉を作るゲームです。古くからある『スクラブル』というボードゲームに似ています。決められた個数以上の言葉を作ると、ステージクリアとなります。このゲームの特徴は、膨大な言葉の辞書を用意していることです。意外な言葉ができたり、言葉の意味が表示されたりするので、見ているだけでも楽しいゲームです。

■このeたこ（このいーたこ）

・ミッチェル：2003：AC, PC

・ブロックを引き寄せて撃つ

上下に動くタコを操作して、左右から迫ってくるドラム缶を、次々に消していくゲームです。タコはドラム缶を吸い寄せたり、撃ち返したりすることができます。同じ種類のドラム缶を隣接させると、消すことができます。撃って積むときだけではなく、吸って引き抜くときにも、ドラム缶の配置が変わるので、消すチャンスがあります。また、片側から吸い寄せたドラム缶を、反対側に撃ち込むことができる点もユニークです。

■コラムス

・セガ：1990：AC, DC, GBA, PS2, SS, WII

・次のブロックを表示する、宝石を落とす、宝石の順番を変える、縦横斜めに揃える

ステージ上方から降ってくる宝石を積み上げる、落ち物パズルゲームです。同じ種類の宝石を、縦・横・斜めのいずれかに3個並べると、消すことができます。宝石を消すことで、上にあった宝石が落下して、連鎖的に消えることがあります。まれに魔宝石と呼ばれる特別な宝石が降ってきて、この宝石を落とすと、同じ色の宝石をまとめて消すことができます。

■XI（さい）

・ソニー・コンピュータエンタテインメント：1998：PS, PS2, PSP

・サイコロを揃えて消す

ステージ上に出現するサイコロを、次々に消していくゲームです。キャラクターはサイコロの上に乗って転がしたり、サイコロを横から押したりすることができます。サイコロの上面に出ている目の数だけ、同じ目のサイコロを隣接させると、サイコロは消えます。サイコロを転がすと、サイコロの位置と目が同時に変化するので、次にどんな状態になるのかをよく考えながら、素早く操作する必要があります。

■さめがめ

・個人作品：1985：PC

・つながったブロックを消す

つながったブロックを、できるだけ多くまとめて消すゲームです。シンプルなゲームながら、最初のブロックの配置や、ブロックを消す順番によって、大きくスコアが変わるため、ついつい何度もプレイしてしまう中毒性があります。原作は森辺訓章（もりすけ）氏によるフリーソフトウェアです。ルールやソースコードが公開されているため、数々の移植作品があります。

■シンドバッドミステリー

・セガ：1983：AC
・地図を頼りにアイテムを探す

地図を頼りに、敵をかわしながら、ステージ内に隠された財宝を見つけるゲームです。ステージには地図の断片が散らばっています。断片を拾うたびに、少しずつ地図が見えるようになります。地図と地形を見比べて、財宝の場所を推理するのが楽しいゲームです。階段や橋など、2次元ながらも立体感のあるステージも特徴です。

■スーパーパズルファイターⅡX（スーパーパズルファイターつーえっくす）

・カプコン：1996：AC, DC, PC, PS, SS
・エサのブロックを消す

『ストリートファイター』シリーズや『ヴァンパイア』シリーズのキャラクターをモチーフにした、落ち物パズルゲームです。ステージ上方から降ってくる「ジェム」と呼ばれる宝石を、左右に動かしたり、回転させたりしながら、積み上げます。ときどき降ってくる「クラッシュジェム」と呼ばれる特別な宝石を落とすと、上下左右に隣接した同じ色の宝石を壊して、消すことができます。宝石の消し方は『ばくばくアニマル』に似ています。

■戦球（せんきゅう）

・セイブ開発：1995：AC, PS
・ぶら下がったボール

ボールを落として積み上げる、落ち物パズルゲームの一種です。同じ種類のボールを4個以上隣接させると、ボールを消すことができます。このゲームの特徴は、ボールが互い違いに積み上がることです。ボールの配置は、『パズルボブル』のぶら下がったボールに似ています。新しいボールを落としたときに、積み上げたボールの段差に応じて、左右に転がっていく動きがユニークです。

■線脳（せんのう）

・カネコ：1999：AC
・パイプをつなぐ

『カチャット』に似たゲームです。ステージ上方から降ってくるパイプを、左右に動かしたり回転させたりして、積み上げます。パイプをつなげてループを作ると、消すことができます。ループにつながったパイプだけではなく、つながっていないパイプも、ループで囲むと一緒に消すことができるため、余分なパイプを処分することが容易になっています。

■倉庫番（そうこばん）

・シンキングラビット：1982：AC, PC, PS
・迷路を歩く、荷物を押して運ぶ、荷物を指定の場所に運ぶ

荷物を押して目的地まで運ぶゲームです。荷物を引くことはできないので、荷物を運ぶ経路や順番をよく考えないと、容易に行き詰まってしまいます。シンプルなゲームですが、ステージの構成がよく練られていて、あきさせません。かつては数々のプラットフォームに移植され、現在も携帯電話などで遊べます。

■ソロモンの鍵（ソロモンのかぎ）

・テクモ：1986：AC, PS2, WII, XB
・床を作って進む

敵を避けながら、アイテムを回収し、ステージから脱出するゲームです。「換石の術」と呼ばれる技を使って、

キャラクターは床を作ったり壊したりします。床を足場にして高いところに登ったり、壁を壊して抜け穴を作ったりすることができます。また、壁で囲んで敵を閉じ込めたり、敵の足下の床を壊して落としたりすることも可能です。

■対戦とっかえだま（たいせんとっかえだま）

・コナミ：1996：AC, PS
・ボールを入れ替える

『対戦ぱずるだま』に似たゲームですが、ボールを落として積むのではなく、ボールを入れ替える方式になっています。同じ種類のボールを3個以上隣接させると、消すことができます。ステージ下方から、時間とともにボールがせり上がってくるので、ステージがボールで埋まらないように、次々に消していきます。ボールを連鎖的に消すと、対戦相手にボールを送り込むことができます。

■対戦ぱずるだま（たいせんぱずるだま）

・コナミ：1994：AC, GBA, PC, PS, PS2, SS
・次のブロックを表示する、一度では消えないボール

ステージ上方から降ってくるボールを積み上げる、落ち物パズルゲームです。同じ種類のボールを3個以上隣接させると、消すことができます。『ぷよぷよ』に似ていますが、普通に消せる「おおだま」と、隣接したおおだまを消すことによって、おおだまに変化する「こだま」がある点が違います。また、ボールを消すと相手にボールを送ることができますが、このボールは消しやすい規則的な配置をしているため、送られたボールを利用した反撃が可能です。

■ダンシングアイ

・ナムコ：1996：AC
・線で囲む

『クイックス』の3次元版のようなゲームです。敵をかわしながら、キャラクターを動かして線を引き、領域を囲みます。本作品の特徴は、画面の中心に3Dグラフィックの女性キャラクターが表示されており、その周囲にフィールド（服）が配置されている点です。囲んだ領域は消えるので、隠されていたキャラクターがだんだん見えてきます。ゲームの内容としては『ギャルズパニック』の3次元版といったところで、やはり独特の鬼気迫る雰囲気があります。

■チェックマン

・ジャレコ：1982：AC
・一筆書きでアイテムを回収する

キャラクターを動かして、敵や障害物を避けながら、アイテムを回収するゲームです。キャラクターが通ると軌跡が残ります。軌跡を再び通過することはできないため、一筆書きの要領でアイテムを集める必要があります。画面の両端がつながっているので、経路を工夫すれば、取れそうにないアイテムを回収できることが意外によくあります。

■テトリス

・セガ：1988：AC, DC, DS, GBA, GC, PS, PS2, SS, XB, XB360
・ブロックを落とす、ブロックを左右に移動する、ブロックを回転させる、ブロックを一段揃えて消す、次のブロックを表示する

落ち物パズルゲームの代表作です。ステージ上方から落ちてくるいろいろな形状のブロックを、左右に移動さ

せたり回転させたりして、隙間なく積み上げることが目的です。ブロックを横1段に隙間なく揃えると、その段を消すことができます。元は旧ソビエト連邦の科学者が開発したゲームですが、非常に多くのプラットフォームに移植され、広く遊ばれています。

■テトリス ザ・グランドマスター

・アリカ：1998：AC, XB360

・落下予測位置を表示する、障害物を避けながらボールを回転させる

数多くある『テトリス』の関連作品のなかで、最も難易度が高いといわれている作品です。基本的なルールは『テトリス』のものを継承しながら、上級者向けの独自機能がいろいろと追加されています。レベルが上がるとブロックが一瞬で着地するなど、高い難易度を追求する一方で、落下予測位置を表示したり、出現前にブロックを回転させられたりと、快適に遊ぶための工夫も盛り込まれています。

■ドリームショッパー

・サンリツ：1982：AC

・アイテムの位置を記憶する

キャラクターを動かしてアイテム（パネル）を回収し、スコアを稼ぐゲームです。一定以上のスコアを稼ぐと、ステージクリアとなります。スコアはアイテムごとに異なるため、よりスコアの高いアイテムを回収することがポイントです。アイテムはステージ開始時に短時間だけ表示され、ステージが始まると見えなくなるので、高いスコアのアイテムがどこにあるのかを記憶しておく必要があります。

■パイプドリーム

・ビデオシステム：1990：AC

・パイプをつなぐ

スタートからゴールまでパイプをつないで、液体を運ぶゲームです。ステージ開始から一定時間が経過すると、スタートから液体が流れ出します。液体がパイプの先からこぼれる前に、素早くパイプをゴールまでつなぐ必要があります。使うパイプの本数の下限が決められているので、ステージの形状を考えつつ、十字のパイプなどを駆使して、できるだけ長いルートを作らなければなりません。

■ばくばくアニマル

・セガ：1995：AC, PC, SS

・エサのブロックを消す

動物とエサのブロックが降ってくる、落ち物パズルゲームです。イヌ・ウサギ・サル・パンダなどの動物のブロックが、それぞれ骨・にんじん・バナナ・笹といったエサのブロックを、食べて消します。動物とエサが隣接すると、動物がバクバクとエサを食べる演出があるなど、ほのぼのとした楽しいゲームです。

■パズループ

・ミッチェル：2001：AC, DS

・軌道に沿って進むボール、新しいボールを軌道上に追加する、ボールを任意の方向に撃つ、ボールを軌道に撃ち込む、軌道上に並んだボールを消す

軌道に沿って進むボールを消すゲームです。左右に回転する砲台を使って、軌道に新しいボールを撃ち込みます。軌道上で同じ種類のボールを3個以上並べると、消すことができます。ボールが軌道の末端に達する前に、次々にボールを消すことがゲームの目的です。ボールを消したときに、消えた部分の両端が同じ種類のボール

ならば、ボールが引き寄せられて後方に戻ります。この性質を使って、ボールを後方に戻しながら消すことがポイントです。

■パズルスター

・アイジーエス：1999：AC

・ブロックを突き落として集める

ステージに積み上げられたブロックを、突き落として別の領域に送り、同じ種類のブロックを集めて消すゲームです。キャラクターを上下に動かして、ボタンを押すと、キャラクターと同じ段のブロックを突き落とすことができます。ブロックを突き落とすと、上にあったブロックが落ちてくるため、突き落とす順番をよく考える必要があります。

■パズルボブル

・タイトー：1994：AC, DC, GC, PC, PS, PS2, SS, XB

・ぶら下がったボール、撃ったボールが跳ね返る、撃ったボールがぶら下がる、ぶら下がった同じ種類のボールを消す、ボールの軌道を予測して表示する

ステージ上方にぶら下がったボール（泡）を消すゲームです。左右に回転する砲台を使って、新しいボールを撃ち込むと、新しいボールがぶら下がります。同じ種類のボールを3個以上隣接させると、ボールを消すことができます。撃ったボールは壁で反射します。反射を上手に使って、ぶどうの房のようにぶら下がったボールの根元を狙って消すと、多くのボールを一気に落として消すことができます。

■ピクミン

・任天堂：2001：GC

・自律的に動くキャラクター

ピクミンというキャラクターに命令を与えて、宇宙船のパーツを集めるゲームです。ピクミンを育てたり、敵を倒したり、アイテムを回収したりといったアクションは、RTS（Real-Time Strategy）的です。秀逸なのは操作方法で、笛を吹いてピクミンを集合させたり、ピクミンを拾って投げたりと、非常にシンプルで直観的なユーザインタフェースを採用しています。

■ピッキン

・バラドンオートメーション：1983：AC

・荷物を指定の場所に運ぶ

迷路のなかに配置された荷物を、目的地に運ぶゲームです。荷物には何種類かの色があり、目的地も色別に用意されています。キャラクターを動かし、荷物を拾って、対応する目的地まで運びます。他の荷物のある場所は、荷物を持って通ることができないので、荷物を運ぶ順番や経路をよく考える必要があります。また、動き回っている敵が、ときどき勝手に荷物を運んでしまいます。

■フォゾン

・ナムコ：1983：AC, PC, PS

・結合して形を作る

化学反応をモチーフにしたゲームです。「ケミック」と呼ばれる自機を動かして、浮遊する「モレック」と呼ばれる物体と結合し、目的の形を作ります。モレックが結合する位置は、接触した位置によって変わります。多数のモレックが結合するにつれて、目的の位置に結合させるのが難しくなることが、このゲームの面白いところ

です。モレックが間違った位置に結合してしまったら、ボタンを押して切り離すこともできます。

■ぷよぷよ

・コンパイル/セガ：1992：AC, DC, DS, GBA, PC, PS, PS2, PSP, SS, WII, XB

・次のブロックを表示する、ボールを落とす、障害物を避けながらボールを回転させる、着地したボールが2つに分かれる、連鎖的に消す、相手側にボールを降らせる

「ぷよ」と呼ばれるボールを積み上げる落ち物パズルゲームです。同じ種類のぷよを4個以上隣接させると、消すことができます。ぷよが消えると、上にあったぷよが落下し、連鎖的に消えることがあります。大きな連鎖を組むことで、対戦相手に大量の「おじゃまぷよ」と呼ばれる透明なボールを送り込むことができます。積極的に連鎖を組んでいくプレイスタイルの、爽快感があるゲームです。

■フラッピー

・デービーソフト：1983：PC, WII

・迷路を歩く、重力で落ちる荷物を押して運ぶ、荷物を指定の場所に運ぶ

「ブルーストーン」という岩を押して、「ブルーエリア」という目的地に運ぶゲームです。『倉庫番』に似ていますが、岩が重力で下に落ちる点が違います。ブルーストーンの他に、「ブラウンストーン」という岩があり、こちらは敵を倒したり、足場を作ったりするために使います。かわいいキャラクターや敵のデザインも魅力のゲームです。

■ブロックアウト

・テクノス：1989：AC

・ブロックを落とす、3次元のブロックを落とす

『テトリス』の3次元版のような落ち物パズルゲームです。ステージの手前から奥に向かって、3次元のブロックが落ちていきます。ブロックを1段に隙間なく並べると、その段を消すことができます。ブロックの回転方向がX軸・Y軸・Z軸の3種類あるので、思った方向に回転させるのには慣れが必要です。また、積み上げたブロックの奥行きを把握しやすいように、段ごとにブロックが色分けされています。

■ヘクシオン

・コナミ：1992：AC

・ブロックを落とす

『テトリス』に似たゲームですが、六角形のブロックを使っている点が違います。遊び方は『テトリス』とほぼ同じで、ブロックを左右に動かしたり、回転させたりしながら、隙間なく積み上げます。ブロックを隙間なく横1段に並べると、その段を消すことができます。

■ペンゴ

・セガ：1982：AC, PC, SS

・滑る荷物を押す

ペンギンを操作し、氷を使って敵を倒すゲームです。ステージには氷が迷路状に配置されています。氷に接触してボタンを押すと、氷が滑ります。滑った氷で敵を押しつぶすと、倒すことができます。また、ダイヤモンドが描かれた特殊なブロックがステージに3個あり、これらのブロックを1列に並べると、ボーナスが入ります。

■マジカルドロップ

・データイースト：1995：AC, PS, SS, WII

・ボールを拾って集める

ステージ上方から降りてくる「ドロップ」と呼ばれるボールを、次々に消していくゲームです。キャラクターを動かして、頭上にあるドロップを拾ったり、逆に戻したりします。同じ種類のドロップを3個以上縦に並べると、消すことができます。ドロップが消えたときに、動いたドロップが連鎖的に消えるのとは別に、ドロップが消える途中に別のドロップを消したときにも、連鎖消し扱いになります。そのため、あらかじめ連鎖を組むのとは違った、スピード感のある連鎖消しも楽しめます。

■もうぢゃ

・エトナ：1996：AC, PS, SS

・ボールを落とす

『ぷよぷよ』に似た落ち物パズルゲームです。ボールのかわりにコインを落とすことが特徴で、コインを上位のコインに両替することによって消します。コインの種類は1円・5円・10円・50円・100円・500円です。1円・10円・100円は5個以上を、5円・50円は2個以上を隣接させると、上位のコインに変わります。500円を2個以上隣接させると1000円札に変わり、消すことができます。

■マネーアイドルエクスチェンジャー

・フェイス：1997：AC, PC, PS

・ボールを拾って集める

『マジカルドロップ』と『もうぢゃ』を組み合わせたようなゲームです。ボールを拾って集めるかわりに、コインを拾って集めます。コインには1円・5円・10円・50円・100円・500円があり、1円・10円・100円は5個以上を、5円・50円は2個以上を隣接させると、上位のコインに変わります。500円を2個以上隣接させると、コインを消すことができます。女の子のキャラクターが多数登場することも特徴です。

■レミングス

・シグノシス：AC, PC, PS3, PSP

・自律的に動くキャラクター

多数のレミングというキャラクターを、ゴールまで導くゲームです。レミングは自律的に動きます。ステージにはさまざまな罠が仕掛けられているので、タイミングよく適切な命令を与えないと、レミングは次々に死んでしまいます。ステージによって使える命令の種類や回数が違うので、戦略を練る楽しみがあります。

■ロットロット

・アイレム：1985：AC

・遅れて追随するカーソル、転がる大量のボール

転がってくる大量のボールを、ゴールまで導くゲームです。ステージは複数の区画に分かれています。遅れて追随するカーソルが、別の区画を指しているときにボタンを押すと、ある区画と別の区画のボールを入れ替えることができます。ゴールは複数あり、それぞれスコアが違います。ボールの入れ替えを上手に行い、より高いスコアのゴールにボールを導くことが、ゲームの目的です。

索引　Appendix 03

■さ行

■た行

■な行

■は行

■ま行

■や行

■ら行

■サンプル

■引用ゲーム

■クラス

おわりに

「パズルゲームを遊ぶ人は、一度くらい『自分でパズルゲームを作ろう』と思ったことがあるはず！」という考えのもとに、普通のゲームプログラミング入門書とは違った切り口で進めてきた本書でしたが、お楽しみいただけたでしょうか。本書がパズルゲームに関してあれこれ考えたり、楽しく遊んだり、熱く語ったり、プログラムを書いたり、いっそうパズルゲームが好きになったりするための手助けになれば幸いです。

かつて『テトリス』『コラムス』『ぷよぷよ』といった落ち物パズルゲームの全盛時代がありました。その後ブームは一段落し、ゲームの一ジャンルとして落ち着いた感がありますが、今再びパズルゲームが静かなブームを迎えている様子です。じっくりと腰を据えて遊ぶおおがかりなゲームもよいのですが、ちょっとした空き時間や移動時間に、携帯ゲーム機や携帯電話で遊ぶゲームとして、パズルゲームはぴったりです。Webブラウザ上で遊ぶFlashなどをベースにしたゲームとしても、パズルゲームは人気を集めています。これから一段と、パズルゲームを遊んだり、作ったりする機会は増えるのではないでしょうか。

最後に、本書の関連書籍を紹介させていただきます（いずれもソフトバンク クリエイティブ刊）。本書とあわせてお読みいただけば、ゲームプログラミングの世界をより深く楽しんでいただく助けになるかと存じます。ぜひお試しください。

『アクションゲームアルゴリズムマニアックス』

古今東西のさまざまなアクションゲームにおけるギミックを解説した、本書の姉妹書籍です。本書と同じく図解を中心にしているので、漫画を読むような気楽な気持ちで、アクションゲームの世界を堪能していただけます。

『シューティングゲームアルゴリズムマニアックス』

数々のシューティングゲームにおける多種多様なギミックを解説した、本書の姉妹書籍です。やはり図解を中心にしているので、漫画や画集を眺めるようにリラックスして、シューティングゲームの世界を味わっていただけます。

『ゲームエフェクトマニアックス』

数々の3Dエフェクトを作成する方法を満載した書籍です。3Dグラフィックやシェーダープログラミングに興味をお持ちでしたら、カッコいいゲームを作るためにきっと役立てていただけるでしょう。

『シューティングゲームプログラミング』

シューティングゲームの制作方法を実践的かつ平易に解説した書籍です。これからシューティングゲームを作ろうという方にはもちろん、シューティングゲームを題材にプログラミングを学ぼうという方にもお使いいただけます。ゲームライブラリやタスクシステムに関する詳細な解説もあります。

■著者プロフィール

松浦 健一郎(まつうら けんいちろう)
東京大学工学系研究科電子工学専攻修士課程を修了後、研究所勤務を経て、現在は趣味と実益を兼ねつつフリーのプログラマ&ライターとして活動中。著書（いずれも司ゆきと共著）に『はじめてのJBuilder4』『Delphi DB&Webプログラミング』『はじめてのJBuilder6』『デスクトップマスコットを作ろう!!』『シューティングゲーム アルゴリズム マニアックス』『ゲームエフェクト マニアックス』『Javaのココロ』『シューティングゲーム プログラミング』『アクションゲーム アルゴリズム マニアックス』『3D格闘ゲーム プログラミング』『Visual C++ 2005 実用サンプルプログラム Windowsプログラミング Tips 108』（以上、ソフトバンククリエイティブ刊）、『Windows Vistaガジェットプログラミング』『うぇぶマスコットをつくろう!!－for Ajax』（秀和システム刊）がある。関心と仕事の範囲はプログラミングを中心にコンピュータ全般に及ぶが、最も興味がある分野はプログラミング言語作りとゲーム作り。

司 ゆき（つかさ ゆき）
東京大学理学系研究科情報科学専攻修士課程修了。学部生時代からライターおよびプログラマの仕事を始める。庭でシイタケを栽培することを夢見ていたら、いつの間にか巨大なキノコが自生してきて驚いている。

著者Webサイト「ひぐぺん工房」
http://cgi32.plala.or.jp/higpen/gate.shtml

パズルゲーム アルゴリズム マニアックス

2008年7月31日　初版第1刷発行

著　者・・・・・・・・・松浦 健一郎（まつうら けんいちろう）／司 ゆき（つかさ）

発行者・・・・・・・・・新田 光敏
発行所・・・・・・・・・ソフトバンククリエイティブ株式会社
〒107-0052　東京都港区赤坂4-13-13
TEL 03-5549-1201（販売）
http://www.sbcr.jp

印　刷・・・・・・・・・株式会社シナノ
装丁/本文デザイン/組版・クニメディア株式会社

落丁本、乱丁本は小社販売局にてお取り替えいたします。
定価はカバーに記載されております。

Printed In Japan　　ISBN978-4-7973-4709-8

パズルゲーム アルゴリズム マニアックス

COMPACT disc

SoftBank Creative

PuzzleGame Algorithm Maniax